(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
8 February 2001 (08.02.2001)

PCT

(10) International Publication Number
WO 01/10043 A1

(51) International Patent Classification⁷: H03M 13/35, 13/47, G06F 11/10, H04L 1/00

(21) International Application Number: PCT/US00/20398

(22) International Filing Date: 26 July 2000 (26.07.2000)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
09/366,222 2 August 1999 (02.08.1999) US

(71) Applicant: PARADOX DEVELOPMENT CORPORATION, D.B.A. LAND-5 CORPORATION [US/US]; 9747 Business Park Avenue, San Diego, CA 92131 (US).

(71) Applicant and
(72) Inventor: WIENCKO, Joseph, A., Jr. [US/US]; 1105 Robin Road, Blackburg, VA 24060 (US).

(72) Inventors: LAND, Kris; 12277 Berea Court, Poway, CA 92064 (US). DICKSON, Lawrence, J.; 19991 Mcain Lane, Boulevard, CA 91905 (US).

(74) Agent: MEADOR, Terrance, A.; Gray Cary Ware & Freidenrich, LLP, 401 B Street, Suite 1700, San Diego, CA 92101-4297 (US).

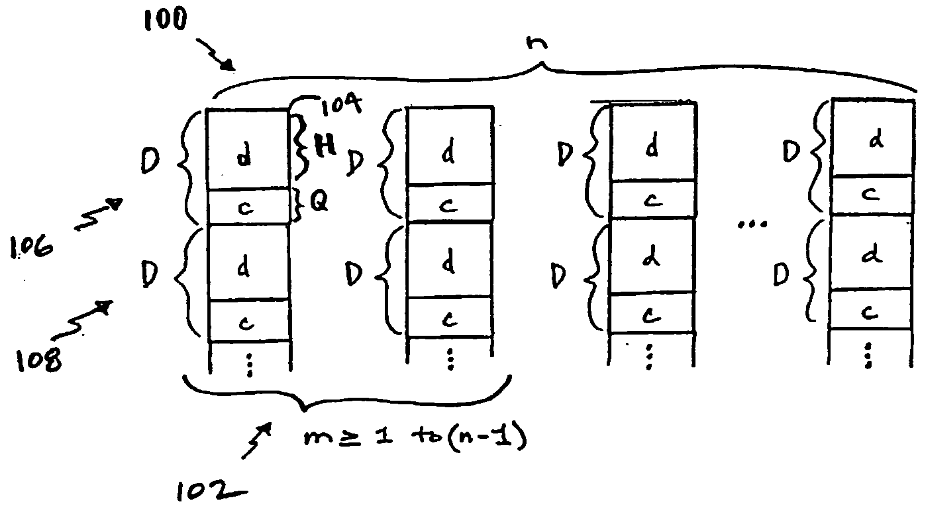
(81) Designated States (national): AE, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CR, CU, CZ, DE, DK, DM, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, UZ, VN, YU, ZA, ZW.

(84) Designated States (regional): ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published:
— *With international search report.*
— *Before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments.*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: DATA REDUNDANCY METHODS AND APPARATUS



(57) Abstract: A data storage apparatus has a plurality of n disks and data comprising a plurality of n data groupings stored respectively across the plurality of n disks. Each one of the n data groupings comprise a data portion and a data redundancy portion. Advantageously, the n data portions are recoverable from any and all combinations of n-m data grouping(s) on n-m disk(s) when the other m data grouping(s) are unavailable, where 1 < m< n. Application of the coding is also applied to rf transmission and a redundant server configuration.

WO 01/10043 A1

DATA REDUNDANCY METHODS AND APPARATUS

REFERENCE TO MICROFICHE APPENDICES

Filed herewith are APPENDICES A (pages A1-A45), B (pages B1-B11), C (pages C1-C10), and D (pages D1-D7). When APPENDICES A, B, C, and D are provided on microfiche, the number of microfiche sheets is _______ and the number of frames associated therewith is _______ .

The microfiche appendices contain material which is subject to copyright protection. The copyright owner has no objection to the reproduction of such material as it appears in the files of the Patent and Trademark Office, but otherwise reserves all copyright rights whatsoever.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates generally to data redundancy methods and apparatus. Various aspects relate more particularly to redundancy data generation, data restoration, data storage, redundancy adjustability, data communication, computer network operations, and code discovery techniques.

2. Description of the Related Art

With the explosive growth in the Internet and mission-critical applications, the importance of preserving data integrity and ensuring 24x7 continuous access to critical information cannot be overstated. Information is now recognized as a key organizational asset, essential to its operation and market competitiveness. Access to critical information on a continuous basis is a mandatory requirement for survival in the business world. Critical applications involving military operations, communications, audio-visual, medical diagnoses, ISP (Internet Service Provider) and Web sites, or financial activities, for example, depend upon the continuous availability of essential data.

Downtime is extremely costly. Customers, vendors, employees, and prospects can no longer conduct essential business or critical operations. There is a "lost opportunity" cost to storage failures as well in terms of business lost to competitors. Well-documented studies place the cost of downtime in the tens of thousands (or even millions) of dollars per hour.

The need for large amounts of reliable online storage is fueling demand for fault-tolerant technology. According to International Data Corporation, the market for disk storage systems last year grew by 12 percent, topping $27 billion. More telling than that figure, however, is the growth in capacity being shipped, which grew 103 percent in 1998. Much of this explosive growth can be attributed to the space-eating demands of endeavors such as year 2000 testing, installation of data-heavy enterprise resource planning applications and the deployment of widespread Internet access.

Disk drive manufacturers publish Mean Time Between Failure (MTBF) figures as high as 800,000 hours (91 years). However, the claims are mostly unrealistic when examined. The actual practical life of a disk drive is 5 to 7 years of continuous use. Many Information Technology managers are aware that disk drives fail with great frequency. This is the most likely reason why companies place emphasis on periodic storage backup, and why there is such a large market for tape systems.

The industry answer to help satisfy these needs has been the use of conventional RAID ("Redundant Arrays of Inexpensive Disks") storage. In general, RAID storage reduces the risk of data loss by either replicating critical information on separate disk drives, or spreading it over several drives with a means of reconstructing information if a single drive is lost.

There are basically four elements of RAID: 1) *mirroring* data (i.e., creating an exact copy every time information is written to storage), 2) performing checksum calculations (*parity* data), 3) *striping* information in equal-sized pieces across multiple drives, and 4) having a standby *hot spare* should one drive fail. Some methods use a combination of both approaches. RAID storage systems are usually designed with redundant power supplies and the ability to swap out failed drives, power supplies and fans while the system continues to operate. Sophisticated RAID systems even contain redundant controllers to share the workload and provide automatic fail-over capabilities should one malfunction.

Conventional RAID storage configurations have proven to be the best hedge against the possibility of a single drive failure within an array. If more than one drive in a RAID array fails, however, or a service person accidentally removes the wrong drive when attempting to replace a failed drive, the entire RAID storage system becomes inoperable. And the likelihood of multiple drive failures in large disk arrays is significant. The resultant cost of inaccessibility to mission-critical information can be devastating in terms of lost opportunity, lost productivity and lost customers.

Accidents can contribute to multiple drive failures in RAID storage. Service personnel have been known to remove the wrong drive during a replacement operation, crashing an entire RAID storage system. In poorly engineered RAID systems, replacing a failed drive can sometimes create a power glitch, damaging other drives. General data center administrative and service operations also present opportunities for personnel to inadvertently disable a drive.

It is well-known that the likelihood of a drive failure increases as more drives are added to a disk RAID storage system. The larger the RAID storage system (i.e., the more disk drives it has), the greater the chance that two or more drives could become inoperable at one time. Here, the term "time" means the duration from the instant when a drive fails until it is replaced and data/parity information is recovered. In remote locations, during holidays, or even during graveyard shifts, the "time" to drive recovery could be several hours. Thus, multiple drive failures do not have to occur at exactly the same instant in order to have a devastating effect on mission-critical storage.

Given the plausible assumptions that drives fail independently at random times with a certain MTBF, and that they stay down a certain time after failing, the following conclusions may be drawn for large arrays of disks: (1) the frequency of single drive failure increases linearly as the number of disks n; (2) the frequency of two drives failing together (a second failing before the first is reconstructed) increases as n*(n-1), or almost as the square of the number of disks; (3) the frequency of three drives failing together increases as n(n-1)(n-2) or almost as the cube; and so forth.

The multiple failures, though still less frequent than single disk failure, become rapidly more important as the number of disks in a RAID becomes large. The following table illustrates the behavior of one, two and three drive failure MTBFs given that single drive MTBF divided by downtime is very much greater than the number of drives:

# of Drives	1	2	3	4	5	10	15	20
MTBF	a	a/2	a/3	a/4	a/5	a/10	a/15	a/20
MTB2F	--	b	b/3	b/6	b/10	b/45	b/105	b/190
MTB3F	--	--	c	c/4	c/10	c/120	c/455	c/1140

Here a << b << c are mean time constants for a failure of one disk, a coincidental failure of two disks, and a coincidental failure of three disks, respectively. If one-disk MTBF is five 360-day years and downtime is one day, then a = 5 years, b = 4,500 years, and c = 5,400,000 years. If MTBF is reduced to 1 year and downtime increased to two days, then a = 1 year, b = 90 years, and c = 10,800 years.

The consequences of a multiple-drive failure can be devastating. Typically, if more than one drive fails, or a service person accidentally removes the wrong drive when attempting to replace a failed drive, the entire RAID storage system is out of commission. Access to critical information is not possible until the RAID system is re-configured, tested and a backup copy restored. Transactions and information written since the last backup may be lost forever.

Thus, the possibility of a multiple-drive failure is very high for mission-critical applications that run 24-hours daily on a continuous basis. Moreover, the larger a RAID storage system, the greater the potential of suffering multiple-drive failures. And the chances increase significantly for remote locations where the response time to replace a failed drive can extend to several hours or even days.

Conventional RAID levels have their advantages and disadvantages. While RAID-0 delivers high performance, it cannot sustain even a single drive failure because there is no parity information or data redundancy. Although the most costly, mirroring data on separate drives (RAID-1), means that if one drive fails, critical information can still be accessed from the mirrored drive. Typically, RAID-1 involves replicating all data on two separate "stacks" of disk drives on separate SCSI channels, incurring the cost of twice as many disk drives. There is a performance impact as well, since data must be written twice, consuming both RAID system and possibly server resources. RAID-3 and RAID-5 allow continued (albeit degraded) operation by reconstructing lost information "on the fly" through parity checksum calculations. Adding a global hot spare provides the ability to perform a background rebuild of lost data.

With the exception of costly RAID-1 (or combinations of RAID-1 with RAID-0 or RAID-5) configurations, there have been few solutions for recovering from a multiple drive failure within a RAID storage system. Even the exceptions sustain multiple drive failures only under very limited circumstances. For example, a RAID-1 configuration can lose multiple (or all) drives in one mirrored stack as long as not more than one drive fails in its mirrored partner. Combining striping and parity within mirrored stacks buys some additional capabilities, but is still subject to these drive-failure limitations.

Some variations of RAID are based merely on combinations of RAID levels, described below in terms of basic structure and performance (0+1 array, 5+1 array, and 5+5 array). All of the packs described in the following configurations are assumed to have pattern designs that maximize read and write speed for large files and parallel data flows to and from disks. The "ideal" speeds will be based on raw data movement only, ignoring buffering and computational burdens: In a "0+1" array, two striped arrays of five disks each mirror the other. A striped array (RAID-0) is lost if only one of its disks is lost, so the safe loss count = 1 and maximum loss count = 5 (depending whether disks lost are on same side of mirror). Data capacity = 5, read speed = 10 (using an operating system capable of alternating mirror reads to achieve full parallelism; the usual max is 5) and write speed = 5 (here reading assumes a strategy of alternating between sides of the mirror to increase the parallelism). In a "5+1" array, two RAID-5 arrays of five disks each mirror each other. Safe loss count is 3 (when one side has lost no more than one disk, the other perhaps more, we can still recover), max loss count is 6 (one entire side, and one disk from the other side). Data capacity is 4 (equals that of one RAID-5 array), read speed = 10 but usual max is 5 (see above discussion of "0+1"), and write speed = 4 (using full parallelism, but with parity and mirror burdens). Similar results arise from a 1+5 array (a RAID-5 made of mirrored pairs). In a "5+5" array, three RAID-5 arrays of three disks each form a RAID-5 with respect to each other. Thus one entire array of three can be lost, plus one disk of each of the other two. This implies safe loss count = 3 (it can't tolerate a 0 - 2 - 2 loss pattern) and max loss count = 5. Data capacity is 4 (of 9), read speed is 9 (using nested striping) and write speed is 4.

Other RAID-like variations exist, but with their downsides. A highly complex encryption-type multiple redundancy algorithm exists, referred to as the Mariani algorithm (downloaded file "raidzz" and related files). The form of RAID described by Mariani can either be applied to dedicated parity disks or have rotation superimposed (as with the two patents referred to below), and additionally requires encryption, which does not treat every bit in a chunk identically. In addition, the subject matter in U.S. Patent Number 5,271,012 ("Method and Means for Encoding and Rebuilding Data Contents of up to Two Unavailable DASDs in an Array of DASDS" and in U.S. Patent Number 5,333,143 ("Method and Means for B-Adjacent Coding and Rebuilding Data from up to Two Unavailable DASDs in a DASD Array") address multiple failures, but are limited. The form of RAID described in these patents generates two parity stripes as a function of n-2 data stripes. The two parity stripes (on two of the disks) are all parity; the n-2 data stripes (on n-2 of the disks) are all data. This leads to read inefficiency unless a rotation structure is superimposed on the formula, in which case it leads to algorithmic inefficiency.

Accordingly, what is needed are methods and apparatus that overcome these and other deficiencies of the prior art.

SUMMARY OF THE INVENTION

A data storage apparatus has a plurality of n disks and data comprising a plurality of n data groupings stored across the plurality of n disks. Each one of the n data groupings comprise a data portion and a redundancy portion. Advantageously, the n data portions are recoverable from any and all combinations of n-m data grouping(s) on n-m disk(s) when the other m data grouping(s) are unavailable, where $1 < m < n$.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is an illustration of a disk array configured in accordance with principles of the invention, where n is the number of disks and m is the number of failures allowed.

FIG. 2 is an example of a conventional RAID configuration.

FIG. 3 is an example of an inventive configuration comparable to the conventional RAID configuration of FIG. 2.

FIG. 4 is an illustration of multiple failure conditions of a storage system from which data is recoverable.

FIG. 5 is an illustration of multiple failure conditions of a storage system from which data is recoverable.

FIG. 6 is a diagram of hardware which may embody principles of the present invention.

FIG. 7 is an illustration of various software components which may embody principles of the present invention.

FIG. 8 is an example of a disk array where m (the number of disks) is five and n (the maximum number of failures allowed) is two.

FIG. 9 shows an exemplary redundancy data generation matrix and functions for m=2 and n=5.

FIG. 10 shows an exemplary data recovery matrix and functions for m=2 and n=5, for data on disks A and B being unavailable.

FIG. 11 shows an exemplary data recovery matrix and functions for m=2 and n=5, for data on disks A and C being unavailable.

FIG. 12 shows an exemplary data recovery matrix and functions for m=2 and n=5, for data on disks A and D being unavailable.

FIG. 13 shows an exemplary data recovery matrix and functions for m=2 and n=5, for data on disks A and D being unavailable.

FIG. 14 is a diagram of a computer system having a redundancy data generator and data recovery component.

FIG. 15 is a schematic block diagram of a storage controller operative in connection with a disk array.

FIG. 16.1 is a schematic block diagram of an adjustable redundancy and recovery apparatus.

FIG. 16.2 is a flowchart describing a method of providing data redundancy adjustments.

FIG. 17.1 is a flowchart describing a method of storing data on the disk array.

FIG. 17.2 is a flowchart describing a method of generating redundancy data.

FIG. 18.1 is a flowchart describing a method of providing data from the disk array.

FIG. 18.2 is a flowchart describing a method of recovering data.

FIG. 19 is a block diagram of a communication system, which here provides for radio frequency (RF) communication.

FIG. 20 is a schematic block diagram of a communication device of FIG. 19.

FIG. 21 is a timing/data diagram of the communication system of FIGs. 19 and 20.

FIG. 22 is the timing/data diagram of FIG. 21, where some data portions are unavailable.

FIG. 23 is a timing/data diagram of the communication system of FIGs. 19 and 20.

FIG. 24.1 is the timing/data diagram of FIG. 23, where some data portions are unavailable.

FIG. 24.2 is a flowchart describing a method of communicating data in the communication system of FIGs. 19 and 20.

FIG. 25 is a diagram of a local area network (LAN) in communication with a switch controller.

FIG. 26 is a diagram of a wide area network (WAN) in communication with telephone carriers.

FIG. 27 is a computer network having multiple servers.

FIG. 28 is a first table showing data reception times of the computer network of FIG. 27.

FIG. 29 is a second table showing data reception times of the computer network of FIG. 27.

FIG. 30 is a third table showing data reception times of the computer network of FIG. 27.

FIG. 31 is a flowchart describing a method of data communication of the computer network of FIG. 27.

FIG. 32 is a flowchart describing a method of determining and/or verifying whether a candidate bit matrix is a Wiencko bit matrix.

FIG. 33 is an example of a Wiencko bit matrix.

FIG. 34 is an example of defining subarrays and composite submatrices from a bit matrix where m=2 and n=5.

FIG. 35 is an example of defining subarrays and composite submatrices from a bit matrix where m=3 and n=5.

FIG. 36 is an example of defining subarrays and composite submatrices from a bit matrix where m=4 and n=5.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

Extensive research and development has resulted in software algorithms that may be used to augment RAID storage technology by performing automatic, transparent recovery from multiple drive failures without interrupting ongoing operations. The inventive techniques extend RAID functionality in ways that allow for instantaneous data recovery in the event of multiple simultaneous or near simultaneous disk failures in a disk array.

In accordance with one inventive aspect, a data storage apparatus has a plurality of n disks and data comprising a plurality of n data groupings stored in the plurality of n disks respectively. Each one of the n data groupings comprise a data portion and a data redundancy portion. Advantageously, the n data portions are constructible from any and all combinations of (n-m) data grouping(s) from (n-m) disk(s) when the other m data grouping(s) are unavailable, where $1 < m < n$.

Thus, in an n-drive system, continued operations are possible even in the event of any combination of up to m drive failures (where $1 < m < n$). This adds a ground-breaking element to "extreme uptime" disk array technology, one that may find a natural home in many 24x7 operational environments. Moreover, because these algorithms have exceptionally fast computational speeds, storage transfer rate performance actually increases while adding virtually unlimited data protection. Compared to conventional RAID storage, the inventive storage system improves storage reliability while enhancing overall system performance. The functions are simple and may be easily stored in gate level logic or logic arrays. Preferably, minimal and "sparse" encoding functions are selected to minimize the amount of logic needed to encode the redundancy information. The algorithms may also allow for users to select the degree of "disk-loss insurance" desired.

As an example, an 18-disk array can be "coded" to allow three disks to be missing (n=18, m=3). With an "18/3" configuration, a total of fifteen disks worth of information would be stored on the 18-disk array. 1/18th of the original data is placed on each of the eighteen disks, and an additional twenty percent of coding information is included on each disk. Any three of the eighteen disks may fail and the full data can be reconstructed, in the same way that a single disk of an 18-disk RAID-5 array can fail and the data can be reconstructed.

As described herein, many inventive aspects relate to providing useful and practical applications for a new family of codes, herein referred to as "Wiencko codes" (pronounced "WEN-SCO" codes). The codes may be applied in connection with simple exclusive-OR (XOR) logic. In data storage applications, this is a family of codes where n is the total number of disks in the array, and m is the number of disks that are allowed to be missing while still being able to fully recover the data. When n-m disks survive, the original data can be fully recovered. The efficiency is ideal, because the original quantity of "clear data" would occupy n-m disks. The inventive techniques may extend RAID functionality, while still retaining the advantages of traditional RAID techniques.

In a more particularly described application, a disk array has n disks with a stripe set stored across the n disks. Each stripe of the stripe set has a data portion of H data bits and a redundancy portion of Q redundancy bits. A relationship exists between the data portions and the redundancy portions based on an n*H by n*Q bit matrix. The bit matrix is representible by an n by n array of H by Q bit submatrices, where n/m = (H/Q) + 1. The bit matrix also has a plurality of n!/(m!*(n-m)!) composite bit submatrices definable therefrom. Each such composite bit submatrix is more particularly definable from bit submatrices at the intersection of a unique selection of m column(s) of the n by n array and a unique selection of (n-m) row(s) of the n by n array that correspond to those (n-m) column(s) not included in the unique selection of m column(s). Each one of these composite submatrices is invertible. The relationship between the data portions and the redundancy portions is such that each one of Q redundancy bits is the exclusive-OR of the n*H bits of the data portions and the n*H bits in the row of the bit matrix associated with such redundancy bit.

Data Storage. FIG. 1 is an illustration of a plurality of n devices 100, of which up to m devices 102 may be unavailable and the user data is still recoverable. A number of stripe sets are spread across the n devices 100, such as a plurality of n stripe sets 106 and a plurality of n stripe sets 108. Each one of n devices 100 has a stripe which includes user data (indicated as "d") and redundancy data (indicated as "c"). The user data is in "clear" form and the redundancy data is generated based on Weincko redundancy generation functions.

Referring now to FIG. 4, an illustration shows a number of different failure conditions 400 that a storage system can tolerate using one exemplary configuration of the present invention. The storage system has five disk drives (A through E), where any and all possible combinations of up to two drives may fail and the data is still recoverable. An "X" over a disk drive indicates a failure for that drive. A failure condition 402 shows that disk drives A and B have both failed; a failure condition 404 shows that disk drives A and C have both failed; a failure condition 406 shows that disk drives A and D have both failed; and so on for a failure condition 408 where disk drives D and E have both failed. For all of these failure conditions, the data is still recoverable from the storage system. These failure conditions are more fully summarized in a table 410 in FIG. 4.

FIG. 5 is a diagram illustrating a number of different failure conditions 500 that an inventive storage system can tolerate using another exemplary configuration of the present invention. Here, the storage system has five disk drives (A through E) where any and all possible combinations of up to three drives may fail and the data is still recoverable. A failure condition 502 shows that disk drives A, B, and C have failed; a failure condition 504 shows that disk drives A, B, and D have failed; a failure condition 506 shows that disk drives A, B, and E have failed; and so on for a failure condition 508 where disk drives C, D, and E have failed. For all of these failure conditions, the data is still recoverable from the storage system. These failure conditions are more fully summarized in a table 510 in FIG. 5.

As apparent, the redundancy configuration of FIG. 5 provides more data redundancy than that of FIG. 4. The tradeoff should be apparent: a storage system with more data redundancy (FIG. 5) is less likely to have a nonrecoverable failure but will have less space to store user data. Users of a storage system may want some flexibility to adjust this data redundancy. As will be described in more detail below, the inventive storage system may allow for adjustment of "disk loss insurance" in response to user input (e.g., administrative control data). The failure conditions shown in FIGs. 4 and 5, for example, could be failure conditions from the same storage system in two different programmed configurations. In one embodiment, the variable m can be made adjustable from 1 to (n-1).

Data Storage and Wiencko Codes. Definitions are provided for discussion. These definitions should not be used in a limiting sense when construing terms, but are provided in order to teach those skilled in the art how to practice aspects of the invention in connection with the examples provided.

A "Wiencko system" is a collection of similar stripe sets occupying a set of n parallel devices, having the capabilities of data storage, parity formation, and reconstruction as described below. A "device" may be a physical disk, a partition of a physical disk, a file on a physical disk, or other data storage hardware or software or parallel data communications hardware or software of any sort. A "disk" in the context of this description is the same as device. A "physical disk" may be a hard drive, floppy drive, or other randomly-accessed digital block storage device, or hardware or digital programming of equivalent functionality.

A "stripe" is that part of a stripe set that resides on a single disk. Every stripe of a stripe set is an integer number of chunks, and starts and ends at chunk boundaries. A certain number of chunks of the stripe consist of data, and the rest consist of parity. A "stripe set" is a minimal set of chunks, distributed over all n disks in a Wiencko system, that form a self-contained whole from the point of view of data storage, parity formation, and reconstruction algorithms. The capabilities of a Wiencko system are fully realized over a stripe set. Stripe sets need not influence one another's parity or reconstructed data.

A "block" is the minimum contiguous collection of data that can be read from or written to a device. In the case of a physical disk, a block is often some multiple of 512 bytes (4096 bits); other communication devices may have usable block sizes as small as one bit. A "chunk' is the maximum contiguous collection of data that is treated as an indivisible unit by the Wiencko code. Every bit of a chunk is treated identically, in data and parity operations relating to bits in other chunks at the same relative bit position. A chunk is an integer number of blocks, and starts and ends at block boundaries. A "code stripe" refers to those chunks of a stripe that consist of parity. A "data stripe" refers to those chunks of a stripe that consist of data.

A Wiencko code n.m, where n > m > 0 are integers, may be utilized for storing binary data on n similar devices, in such a way that any m of the devices can be removed or destroyed and all of the data are still recoverable. (Here, "similar" means their raw data storage capability is the same when all organized by one chunk size, but the devices themselves may be physically dissimilar.)

A Wiencko code may be associated with the following terms and relationships. Data is organized in chunks of a certain number of bits each. D chunks form a stripe (wholly on one of the n disks) and n stripes, one on each disk, form a stripe set. D is a certain positive number which is divisible by D1, where D1 is defined to be n/gcd(n,m), and gcd(n,m) is the greatest common divisor of n and m. The integer parameter p is defined to be p = D/D1. Chunk size and stripe size are constant throughout the Wiencko system (at least at one given time), and each disk of the Wiencko system contains the same number of stripes, all organized into stripe sets. For Q = (mD)/n (this is an integer because D1 divides D), and H = D-Q, every stripe consists of H data chunks and Q parity chunks. There is a one-to-one relationship between the raw decoded data stored on the Wiencko system and the data found in the data chunks of all the stripes of all the stripe sets in the Wiencko system.

It follows that the raw data capacity of the Wiencko system equals the theoretically maximum value of n-m, where data capacity is measured in units of one raw disk capacity. It also follows that for a system in which timing is dominated by data transmission timing to and from disks, and data transmission in parallel to or from any set of disks overlaps with no significant time penalty, the Wiencko system approaches the theoretical maximum raw throughput of n for large reads and n-m for "large" writes, where throughput is measured in units of one raw disk throughput and "large" means spanning many stripe sets.

The data in the parity chunks of a stripe set is determined from the data in the data chunks of that stripe set, using a bitwise XOR matrix. Here, "bitwise" means that the k-th bit of any parity chunk is independent of all but the k-th bit of each data chunk in that stripe set (where k is any natural number less than or equal to the chunk size in bits), and the formula for calculating this bit is the same for every k in every stripe set of this Wiencko system. "XOR matrix" means that the formula includes defining a given parity bit by specifying a subset of its available data bits and XORing all the data in those places. Thus, there are n*Q parity formulas, and each parity formula consists of specifying a subset of n*H data bits, one in the k-th position in each chunk.

In addition to being a bitwise XOR matrix, a Wiencko code should be solvable for any combination of m lost disks. A bitwise XOR parity matrix is defined to be solvable for a combination of m disks if the m*H data chunks of a stripe set on those m disks can be uniquely calculated from the (n-m)*Q parity chunks and (n-m)*H data chunks of that stripe set on the remaining n-m disks. For a given n.m, a Wiencko code solution is not necessarily unique for that n.m. For each case, there is a smallest positive integer p for which a Wiencko code exists with parameter p as defined above. A smaller value of p greatly improves the computational efficiency of the algorithm. However, there are cases where it has been found hat no Wiencko code exists where p=1.

A subset of the Wiencko codes, which have advantages in calculation, proving, key storage, and encoding and decoding efficiency, are rotationally symmetric Wiencko codes. These satisfy the following additional condition: if each disk is given a unique index between 0 and n-1, then the parity formula for a given parity (defined by its parity index between 0 and Q-1, and its disk index) specifies its subset from the data chunks (each defined by its data index between 0 and H-1, and its disk index) using only the difference modulo n between the parity disk index and any data disk index, not the absolute data disk index. Thus for a given parity index, the parity formulas are the same for all the disk indices, except for a rotation modulo n of disk indices.

Further details regarding Wiencko codes are now described. In the following discussion, a "bit" shall mean a zero or one, considered as operating under the addition and multiplication rules of the field of order 2. The common name for such addition is "XOR" and for such multiplication is "AND". A "bit vector" shall be a vector whose scalar entries are bits, and a "bit matrix" shall be a matrix whose scalar entries are bits. Operations on bit vectors and bit matrices are analogous to standard vector and matrix operations, following the rules of linear algebra over the field of order 2.

Let H and Q be positive integers, and j and k be positive integers. A "j by k array of H by Q matrices" is a set of j*k matrices, each of dimension H by Q, arranged in double subscript order with the j subscript running fastest (horizontally), where the component matrices themselves have the H subscript running fastest.

This may be visualized as a matrix whose entries are themselves matrices:

$$\begin{array}{cccc}
M_{00} & M_{10} & \cdots & M_{J0} \\
M_{01} & M_{11} & \cdots & M_{J1} \\
\cdots & \cdots & \cdots & \cdots \\
M_{0K} & M_{1K} & \cdots & M_{JK}
\end{array}$$

where J=j-1, and K=k-1. Each M is itself an H by Q matrix.

An "equivalent matrix to an array of matrices" (with dimensions as given above) is the j*H by k*Q matrix that follows by resolving each matrix entry above into its array of scalar entries. Thus, the x+z*H, y+w*Q entry of the equivalent matrix is the x, y entry of M_{zw} in the above array of matrices, where 0<=x<H, 0<=y<Q, 0<=z<j, 0<=w<k.

Let n > m > 0 be integers. A "Wiencko array of matrices of type (n, m)" is an n by n array of H by Q bit matrices, where H and Q are positive integers such that

$$n/m = (H+Q)/Q = H/Q + 1,$$

and the array satisfies the following property: for every set S of m integers between 0 and n-1, the subarray, created from the array by keeping those array matrices whose row index w is not in S and whose column index z is in S, has an invertible equivalent matrix. Here, "invertible" has the usual matrix theoretic meaning as applied to matrices over the field of order two. A common term for this is "parity invertible" and the matrix is a "parity matrix." Note that the process described above creates an m by (n-m) array of H by Q matrices, and by the relationship above, its equivalent matrix is a square matrix.

A "Wiencko matrix" is an equivalent matrix to a Wiencko array of matrices. A "zero diagonal Wiencko array of matrices" is a Wiencko array of matrices that satisfies the additional condition: M_{zw} = 0 for each matrix such that z=w (that is, each diagnal entry). A "rotationally symmetric Wiencko array of matrices" is a Wiencko array of matrices satisfying the following additional condition: $M_{zw} = M_{uv}$ whenever z-w = u-v modulo n. A "zero diagonal, rotationally symmetric Wiencko array standard description" is the subarray of a zero diagonal, rotationally symmetric Wiencko array of matrices created by keeping those entries for which w>0 and z=0. The above conditions make it easy to generate the entire Wiencko array from its standard description. Zero diagonal, and rotational symmetry, and sparseness or near-fullness, are among the conditions on Wiencko matrices that provide more efficient implementations of the algorithm, but are not necessary for functionality.

The following description relates to an example of a Wiencko array. A Wiencko array of type (5,2) with H=3 and Q=2 is shown in FIG. 33. Inspection shows that it is zero diagonal and rotationally symmetric; therefore it can be represented by the standard description

```
010 -
100
100 -
001
010 -
001
000 -
000
```

where the dashes show the bounds of the array entries. It now remains to be proven that it satisfies the "invertibility" conditions. Because of the rotational symmetry, the ten possible combinations (n!/(m!*(n-m)!) = 5!/2!*(5-2)! = 10) found in the subset S of two numbers between 0 and 4 reduce to only two essentially different ones: S = (0, 1) and S = (0, 2).

The S = (0,1) case reduces to those submatrices in the intersection of columns "a" and "b" and the rows "c", "d", and "e":

```
100010                     100010
001100                     011101
010100    Invertible =>    000101
001001                     010101
000010                     000010
```

```
000001                      000001
```

The S = (0, 2) case reduces to those submatrices in the intersection of columns "a" and "c" and rows "b", "d", and "e":

```
010000                      010000
100000                      100000
010010      Invertible =>   000110
001100                      000010
000100                      101000
000001                      000001
```

Since both of these are invertible, so are the other eight (due to the rotational symmetry). Therefore, the bit matrix in FIG. 33 is a Wiencko array.

Referring now to FIG. 32, a flowchart is shown which describes a method for determining and/or validating that a given bit matrix is a Wiencko bit matrix. This method may be embodied in the form of software executable by a processor. Beginning at a start block 3200, data that is representative of candidate bit matrix is received (step 3202). This data represents one possible Wiencko solution for a given m and n. More particularly, the candidate data is representative of an n*H by n*Q candidate bit matrix, which is representible by an n by n array of H by Q bit submatrices, where n/m = (H/Q) + 1. In this method, it may be tested that the n/m = (H/Q) + 1 condition is satisfied.

Referring to FIG. 33, an example of such a candidate bit matrix 3300 is shown for an m=2, n=5, H=3, and Q=2 case. As illustrated, candidate bit matrix 3300 is representible by a 5-by-5 array of 3-by-2 bit submatrices 3302, such as a submatrix 3304.

Back to FIG. 32, an H*m by Q*(n-m) composite submatrix is selected from the candidate bit matrix (step 3204). The composite submatrix is formed by submatrices from the intersection of a unique selection of m column(s) of the array and a unique selection of (n-m) row(s) of the array that correspond to those column(s) not included in the unique selection of m column(s). There are n!/(m!*(n-m)!) such composite submatrices that may be formed from the candidate bit matrix. Referring back to FIG. 33, an example of such a composite submatrix 3206 is shown for the m=2, n=5, H=3, and Q=2 case. As illustrated, composite submatrix 3206 is formed by submatrices 3302 taken from the intersection of a unique selection of m (two) columns of the array (columns B and D) and a unique selection of n-m (three) rows of the array that correspond to those n-m (three) columns not included in the unique selection of m (two) columns (columns A, C, and E). FIGs. 34, 35, and 36 show further examples of forming such subarrays and composite submatrices. More particularly, FIG. 34 shows all such composite submatrices for m=2 and n=5; FIG. 35 shows all such composite submatrices for m=3 and n=5; and FIG. 36 shows all such composite submatrices formed for m=4 and n=5.

Back to the flowchart of FIG. 32, the composite submatrix is tested for invertibility (step 3206). If the composite submatrix is invertible at step 3206, it is determined whether all n!/(m!*(n-m)!) composite submatrices associated with the candidate bit matrix have been tested (step 3210). If all such composite submatrices have been tested and are invertible, the overall test is satisfied and the candidate bit matrix is a valid solution (step 3212). If the composite submatrix is not invertible at step 3206, the test fails and the candidate bit matrix is not a solution (step 3208). If any other candidate bit matrices are to be tested (step 3214), the flowchart repeats at step 3202 (the test may have failed at step 3208, or multiple solutions may be desired after completion of step 3212). If no other candidate bit matrices are to be tested at step 3214, the flowchart ends at a finish block 3216.

The above relationships describe Wiencko codes that are "ideal" and preferred codes. It is understood, however, that variations on Wiencko codes may be utilized. For example, codes may be provided such that for every value of μ, $0 < \mu <= m$, a subset of all possible combinations of μ unavailable disks is recoverable. Any available codes that provide recovery for a limited subset of all combinations of m failures (perhaps a majority) may be utilized.

Reference is now made to FIG. 8. Some of the relationships and examples described above may be repeated in the following description and examples for further clarity. The data and code layout for a disk array with a "5/2" configuration is shown in FIG. 8. Assume a chunk size is one bit. To begin the procedure, a set of operations must be performed on stripes of data. As described above, the number of bits in minimum stripe on a disk depends on a number of factors, including n, m, and a matrix expansion coefficient p that (in many cases) is one. Another contributor is the greatest common factor of the two numbers n and m, which is represented by *GCF*(n, m). The number of bits in a minimum data stripe is $H = p^*(n-m)/GCF(n, m)$ per disk. The number of bits in a minimum code stripe is $Q=p^*m/GCF(n, m)$ per disk. For a "4/2" configuration, with four (4) total disks coded to allow two to be missing, a solution exists in which p=1, and *GCF*(4, 2)=2, so H=1 and Q=1. For a "5/2", n=5, m=2, p=1, *GCF*(5, 2)=1, so H=3 and Q=2. For a "5/3" configuration, n=5, m=3, p=1, *GCF*(5, 3)=1, so H=2 and Q=3.

The user data bits may be stored in "clear" form, so when writing data, the only additional task is to calculate and store the code bits on each disk. Each bit of the code stripe of a given disk is an XOR of some of the data bits of the other disks. An encoding mask r can be defined, composed of mask elements r_{nm} in which a mask element value of "1" means inclusion in the XOR calculation, and a value of "0" means exclusion from the XOR calculation. Suppose n=4 and m=2. The encoding matrix for data bits A, B, C, and D to yield code bits A', B', C', and D' can be represented as follows, with "(+)" denoting "exclusive OR":

$$A' = r_{1,1} A(+)r_{1,2^B}B(+)r_{1,3}C(+)r_{1,4}D$$

$$B' = r_{2,1}A(+)r_{2,2}B(+)r_{2,3}C(+)r_{2,4}D$$

$$C' = r_{3,1}A(+)r_{3,2}B(+)r_{3,3}C(+)r_{3,4}D$$

$$D' = r_{4,1}A(+)r_{4,2}B(+)r_{4,3}C(+)r_{4,4}D$$

Alternatively, this equation set may be represented simply be referring to the mask, r =

$$\{r_{1,1}\ r_{1,2}\ r_{1,3}\ r_{1,4}\}$$
$$\{r_{2,1}\ r_{2,2}\ r_{2,3}\ r_{2,4}\}$$
$$\{r_{3,1}\ r_{3,2}\ r_{3,3}\ r_{3,4}\}$$
$$\{r_{4,1}\ r_{4,2}\ r_{4,3}\ r_{4,4}\}$$

The mask should be chosen such that a valid decoding solution exists for each combination of channels that can be missing. r_{xx} can be chosen to be 0 in all cases with no loss of information.

Example – "4/2" Configuration. Suppose n=4 and m=2. Let A, B, C, and D denote data bits for each of four disks in an array, and A', B', C' and D' denote coding bits. The encoding functions may be defined as follows:

A'=C (+) D		{ 0 0 1 1 }
B'=A (+) D	equivalent to *r* =	{ 1 0 0 1 }
C'=A (+) B		{ 1 1 0 0 }
D'=B (+) C		{ 0 1 1 0 }

Decoding Functions:

Disks A & B Missing:	A=C(+)C'(+)D'	B=C(+)D'
Disks A & C Missing:	A=D(+)B	C=B(+)D'

Disks A & D Missing:	A=B(+)C'	D=B(+)B'(+)D'
Disks B & C Missing:	C=D(+)A'	B=D(+)A'(+)D'
Disks B & D Missing:	B=A(+)C'	D=C(+)A'
Disks C & D Missing:	C=A(+)A'(+)B'	D=A(+)B'

Just as with the encoding functions, the decoding functions can be specified by use of a binary mask. Note that this encoding matrix mask is a circularly-symmetric matrix. When the mask is chosen with this property, a number of advantages are realized. A circularly-symmetric matrix can be fully described by a small subset of the complete mask. If we know the matrix is circularly symmetric, the complete encoding mask above can be fully specified using:

{1}
{1}
{0}

Note that this submatrix is the entire first column, except for the first element. The first element of the first column will always be zero, because it represents the trivial relation A'=A; it is never necessary to encode a disk's own data bits into a disk's coding bits.

Most cases of disk failures can be represented as rotations of a smaller number of unique failure cases. Thus, for A & B missing, B & C missing, C & D missing, and D & A missing, all of these can be considered rotations of the case in which A & B are missing. Similarly, A & C missing and B & D missing can be considered rotations of the case in which A & C are missing. When decoding, the six cases of two disks missing can be represented as rotations of two unique cases. If the decode logic has this rotational ability built in, it is necessary to consider and store only the decode masks of the two rotationally unique cases.

Example – "5/2" Configuration. See FIG. 8, where n =5 disks total, m=2 disks missing, p=1, resulting in d=3 data bits per disk and c=2 coding bits per disk (15 data bits to the array). As shown, the disks are designated A through E. FIG. 9 is an encode function matrix for redundancy data generation. Three data bits of disk A are designed A1, A2, and A3. The two coding bits are designated as A' and A". Data and coding from the other channels B, C, D, and E are represented similarly. The binary matrices are shown with asterisks for ones and spaces for zeroes. The submatrix corresponding to the case in which disks A & B are missing are circled in the 1st table of FIG. 10. This submatrix appears below in the decoding solution for disks A & B missing.

The coding mask selection is done during the design process, and does not need to be repeated during operation. The encoding function can be hardcoded into an arrangement of logic gates, or stored as a binary array mask that is retrieved to control the function of a logic array or in low-level software in a disk array system. As the number of disks in an array increases beyond four or five, it quickly becomes far more advantageous to use the binary array mask storage method.

(1) Decoding Function for A&B Missing. The following discussion makes reference to FIG. 10 of the drawings. For a particular set of m devices missing, a decoding function exists such that the missing data is equal to a set of XOR operations on the surviving code and data. The decoding function is found from the solution of a set of simultaneous equations in modulo-2 number space using XOR operations.

Mathematically, the decoding function is formed by taking a submatrix of the encoding function mask matrix, changing format to add columns for the coding bits, and then taking the modulo-2 matrix inverse of the subset of the encoding function mask matrix. Identical operations are performed on the subset of the encoding mask matrix and the additional coding bit columns.m A solution set is shown in FIG. 10. The modulo-2 matrix inversions for each unique non-rotational case of channels missing are performed during the design process in order to confirm proper mask selection. The results of these inversions constitute the decode functions, and can be stored as binary array masks, just like with the encoding function. Alternatively, since the decoding masks are easy to calculate from the encoding mask, it is an implementation option for the disk array system to calculate the decoding masks when needed (e.g., in software) from the stored encoding masks.

(2) For the decoding function for A&C missing, see FIG. 11. For convenience, the same encoding matrix is also reprinted. However, a different submatrix is highlighted (circled), to correspond with the case in which disks A&C are missing. A solution set is shown in FIG. 11. For the decoding function for A&D missing, see FIG. 12. For convenience, the same encoding matrix is also reprinted. However, a different submatrix is highlighted, to correspond with the case in which disks A&D are missing. A solution set is shown in FIG. 12.

(3) A&D Missing is a Rotation of A&C Missing. See FIG. 13 of the drawings. Since the encoding matrix was designed to be circularly symmetric, the decoding function matrix mask for disk A & D missing is identical to a circular rotation of the case in which disks A & C are missing. This equivalence is demonstrated by starting to solve for disks A & D missing, and then circularly rotating this solution matrix to show that it is the same solution matrix as the case in which disks A & D are missing. To utilize the "A & C disks missing" solution for this case, recognize that disks A & D being missing is just a circular shift of variables by three positions from the case in which disks A & C are missing. Begin with the solution for disks A & C missing, and circularly rotate the variables three positions forward as follows:

$$A \Rightarrow D, B \Rightarrow E, C \Rightarrow A, D \Rightarrow B, E \Rightarrow C.$$

Note that the cases of missing disks (AB, BC, CD, DE, EA) are all circular rotations of each other, and the cases of missing disks (AC, BD, CE, DA, EB) are all circular rotations of each other. Thus, there are ten total solution cases, but because of circular symmetry only two of them are computationally unique.

The encoding method is simple enough to allow, if desired, coding of different sets of data differently on the same array, depending on the importance of each data set. For example, data with low importance can be coded to allow one disk of an array to fail, data with moderate importance can be coded to allow two disks to fail, and data of high importance can be coded to allow three disks to fail. Alternatively, an entire disk array can have all of the data coded the same way.

Embodying and Applying Wiencko Codes/Functions. Referring now to FIGs. 6 and 7, Wiencko codes/functions (as well as all related methods and apparatus described herein) can be embodied in a number of different ways. For example, the codes/functions described can be implemented in a hardware device 600, such as a programmable gate array (PGA), shown in FIG. 6. In this embodiment, simple exclusive-OR gates, such as an exclusive-OR gate 602, are easily combined in accordance with the desired functions. The codes/functions can also be embodied and implemented using software as indicated in FIG. 7. Such software is embedded or stored on a disk 702 or memory 706, and executable on a computer 704 or a processor 708. For providing redundancy adjustability (described in more detail later below), simple switches (hardware) or mask set selection (software) may be used. Here, the appropriate XOR logic circuits or mask sets are selected based on the control data.

Related Methods And Apparatus. FIGs. 16.2, 17.1, 17.2, 18.1, and 18.2 are flowcharts describing general methods which may be used in connection with the various embodiments. More particularly, FIG. 17.1 is a flowchart describing a method of generating redundancy data. Beginning at a start block 1700, user data is received for storage on a disk array (step 1702). Redundancy data is generated based on the user data and the set of data redundancy functions (step 1704). (If multiple sets of data redundancy functions are made available in the system, the set of data redundancy functions are those selected as described below in relation to FIG. 16.2.) The user data and the generated redundancy data is stored on the disk array (step 1706). The flowchart ends at a finish block 1708, but may repeat for other storage requests.

Step 1704 of FIG. 17.1 may be accomplished in connection with the method described in relation to the flowchart of FIG. 17.2. Beginning at a start block 1710, the data is multiplied modulo 2 by a Wiencko bit matrix (step 1712). More specifically, the stripe set data (n*H chunks) is expressed as a column vector to the right of the Wiencko matrix, and multiplied modulo 2 by the Wiencko matrix to get the stripe set parity (n*Q chunks). As apparent, this algorithm is very efficient (especially when the Wiencko matrix is sparse or nearly full). The flowchart of FIG. 17.2 ends at a finish block 1714.

FIG. 18.1 is a flowchart describing a method of recovering user data. Beginning at a start block 1800, a request for user data is received (step 1802). If all of the disks and data are available (step 1804), the user data is read from the disk array (step 1806). In this situation, data manipulation need not take place since the user data may be stored as "clear" data on the disks. The user data is then provided to the requester (step 1814). The flowchart ends at a finish block 1808, but may be repeated for each request. On the other hand, if some of the data is determined to be unavailable at step 1804, the appropriate data recovery functions associated with the available disks or data are selected (step 1810). The user data is then recovered using the selected data recovery functions, available user data and redundancy data (step 1812). The user data is provided to the user (step 1814), and the flowchart ends at finish block 1808.

Step 1810 of FIG. 18.1 may be accomplished using the method described in the flowchart of FIG. 18.2. Beginning at a start block 1816, the stripe set data found in the n-m remaining disks is arranged as in the encoding algorithm, with zeroes in place of the data found in the missing disks (step 1818). This arranged data is multiplied by that subarray of the Wiencko matrix that corresponds to array row numbers not found in S (rows corresponding to available disks) (step 1820). This step gives all the parity chunks for the stripes of the disks not found in S without the influence of the missing data. These parity chunks created are XOR'd with the parity actually found on the disks not found in S (step 1822); which by the linearity of matrix multiplication gives the non-S parity chunks that would have been produced by applying the Wiencko matrix to a data stripe set that included the missing data on the S disks but was zero on all the non-S disks. The appropriate inverse matrix that exists for S is applied to this parity (step 1824), which yields the missing data. If reconstruction is to be carried out, parity for the disks in S can be created by matrix multiplication by the subarray of the Wiencko matrix that corresponds to row numbers in S.

The mathematical concepts involved in encoding and decoding are now described further by example. The simple example below involves a Wiencko array of type (5,2) with H=3 and Q=2:

```
0 0 0|0 0 0|0 1 0|1 0 0|0 1 0
     |     |     |     |
0 0 0|0 0 0|0 0 1|0 0 1|1 0 0
-----|-----|-----|-----|-----
0 1 0|0 0 0|0 0 0|0 1 0|1 0 0
     |     |     |     |
1 0 0|0 0 0|0 0 0|0 0 1|0 0 1   =  W
-----|-----|-----|-----|-----
1 0 0|0 1 0|0 0 0|0 0 0|0 1 0
     |     |     |     |
0 0 1|1 0 0|0 0 0|0 0 0|0 0 1
-----|-----|-----|-----|-----
0 1 0|1 0 0|0 1 0|0 0 0|0 0 0
     |     |     |     |
0 0 1|0 0 1|1 0 0|0 0 0|0 0 0
-----|-----|-----|-----|-----
0 0 0|0 1 0|1 0 0|0 1 0|0 0 0
     |     |     |     |
0 0 0|0 0 1|0 0 1|1 0 0|0 0 0
```

The missing disks, for simplicity, will be 0 and 1. The S={0,1} case reduces to

```
                 dead data #   live data

                0 0 0|0 0 0#0 1 0|1 0 0|0 1 0
                     |     #     |     |
     dead       0 0 0|0 0 0#0 0 1|0 0 1|1 0 0
                -----|-----#-----|-----|-----
     parity     0 1 0|0 0 0#0 0 0|0 1 0|1 0 0
                     |     #     |     |
                1 0 0|0 0 0#0 0 0|0 0 1|0 0 1      W'' # XS
    ########    #############################  =   #########  =  W
                1 0 0|0 1 0#0 0 0|0 0 0|0 1 0      WS  # W'
                     |     #     |     |
                0 0 1|1 0 0#0 0 0|0 0 0|0 0 1
     live       -----|-----#-----|-----|-----
                0 1 0|1 0 0#0 1 0|0 0 0|0 0 0
     parity          |     #     |     |
                0 0 1|0 0 1#1 0 0|0 0 0|0 0 0
                -----|-----#-----|-----|-----
                0 0 0|0 1 0#1 0 0|0 1 0|0 0 0
                     |     #     |     |
                0 0 0|0 0 1#0 0 1|1 0 0|0 0 0
```

where W'' (dead to dead), XS (live to dead), WS (dead to live) and W'(live to live) are subarrays of the Wiencko array. (The same approach will work for deads that are not together, but is more complicated to visualize.)

If D is a data vector, then we similarly can write

```
         D''
 D   =   ###
         D'
```

where D" is the part on the dead channels and D' on the live channels.

Similarly for a parity vector

```
         P''
 P   =   ###
         P'
```

Then by matrix multiplication the encoding procedure gives

```
(1) P  =  W D
```

or

```
    P''       W'' # XS     D''
(2) ###   =   ##########   ###
    P'        WS  # W'     D'
```

or

```
(3a) P''   =  (W'' D'') + (XS D')

(3b) P'    =  (WS  D'') + (W' D')
```

In the case we are working with (bit matrices), adding and subtracting are the same as "XOR" or "^", and multiplication is the same as "AND" or "&".

NOW SUPPOSE D' AND P' ARE KNOWN BUT D" AND P" ARE LOST, but we are assured that (1) holds true for the known matrix W. We now define

```
P1   =   W' D'
```

and

```
P3   =   P'
```

so P1 can be calculated and P3 read up from live data. Now set

```
P2  =  P3 - P1  =  P3^P1  =  P3 + P1
```

We get from (3b) that

$$(4)\ \ WS\ D'' \ = \ P2$$

and because of the definition of a Wiencko Matrix, there exists a matrix

$$JS \ = \ WS^{-1}$$

whence it follows that

$$(5)\ \ JS\ P2 \ = \ JS\ (WS\ D'') \ = \ (JS\ WS)\ D'' \ = \ D''$$

which recovers the lost data. Another expression for (5) is

$$(6)\ \ D'' \ = \ JS\ (I\ \#\ W')\ \begin{array}{c} P' \\ \#\# \\ D' \end{array}$$

where I is a live parity-sized identity matrix. This holds because

$$(I\ \#\ W')\ \begin{array}{c} P' \\ \#\# \\ D' \end{array} \ = \ (I\ P') + (W'\ D') \ = \ P' + P3 \ = P1 + P3 \ = \ P1\hat{\ }P3$$

We define the preconditioning matrix

$$PC \ = \ (I\ \#\ W')$$

We can then do (6) in two different ways:

$$(6a)\ \ D'' \ = \ (JS\ PC)\ \begin{array}{c} P' \\ \#\# \\ D' \end{array} \qquad \text{(single step)}$$

$$(6b)\ \ D'' \ = \ JS\ (PC\ \begin{array}{c} P' \\ \#\# \\ D' \end{array}) \qquad \text{(two step)}$$

The single step method, which precalculates (JS PC), is better for small n, the two step method is better for large n and sparse Wiencko matrices.

Example:
Matrix multiplication modulo 2 is used. # is not a number, but a separator between dead and live territory.

```
        W                            D             P

                                     0
                                     1
   000000#010100010                  1             1
   000000#001001100                  0             0
   010000#000010100                  1             1
   100000#000001001                  1             1              P''
   ################                  #      =      #       =      ###
   100010#000000010                  0             1              P3
   001100#000000001                  1             0
   010100#010000000                  1             0
   001001#100000000                  0             0
   000010#100010000                  1             0
   000001#001100000                  0             0
                                     1
                                     0
                                     1
```

D'' and P'' are lost. D' and P3=P' are known, as is W.

```
        W'                   D'               P1

   000000010                 0                0
   000000001                 1                1
   010000000                 1                1
   100000000                 0        =       0
   100010000                 1                1
   001100000                 0                1
                             1
                             0
                             1


        P1  ^  P3        P2

        0      1         1
        1      0         1
        1      0    =    1
        0      0         0
        1      0         1
        1      0         1


         WS                     JS

        100010                 100010
        001100                 011101
        010100   inverse =     000101
        001001                 010101
```

```
000010            000010
000001            000001


JS          P2           D''

100010      1            0
011101      1            1
000101      1      =     1
010101      0            0
000010      1            1
000001      1            1
```

And thus, D" is recovered. P" can be recovered by encoding from D" and D'.

<u>Detection of Burst Errors With Use of Wiencko Codes.</u> The following discussion relates to an optional technique for error detection with use of Wiencko coding. This technique is especially applicable to burst errors. In some applications, it may be used as the primary error detection technique without use of any other error detection/correction codes. The methods now described may be similarly embodied in hardware or software as described herein.

The corresponding concepts for a Wiencko array -- a row or column whose constituent entries are H by Q matrices -- are named a "channel row array" or "channel column array" to distinguish from these. In cases where the context is clear, either a channel row array or a channel column array may be called a "channel". A channel, in either case, is numbered from 0 to n-1. The "intersection of a row with a channel" is the intersection of a row with (the equivalent matrix of) a channel column array. It is always wholly contained in one entry matrix of the Wiencko array, and always is a row of H bits. The "intersection of a column with a channel" is the intersection of a column with (the equivalent matrix of) a channel row array. It is always wholly contained in one entry matrix of the Wiencko array, and always is a column of Q bits.

Either of these is called "zero" if all its bits are zero, otherwise it is called "nonzero". A set of bits is called "related" if each bit comes from a different chunk of the same stripe set, and each bit is at the same bit offset from the start of its chunk. The maximum number of bits that can be related is n*(H+Q). A set of related bits is called a "related set", and a maximal set of related bits is a "full related set".

The error detection technique is as follows. At all times, both data and parity are received from all n channels. To detect faulty channels, the Wiencko encoding procedure is applied to the data and the results compared with the parity over a stripe set or a number of stripe sets, or over a portion of a stripe set whose bits have the same relative offsets in their respective chunks. All parity chunks which exhibit a discrepancy are noted.

Definition (error bit and delta bit): The "error bit" at position k in a parity chunk is the XOR of the value read at k of that parity chunk and the value calculated using the Wiencko encoding procedure from the values at k of the corresponding data chunks. The "delta bit" of a certain bit location is the XOR of the correct value of the bit and the value read on a faulty channel.

Definition (zero favor): Let R be a random variable that takes the values 0 and 1, 0 with probability p and 1 with probability q = 1-p. The "zero favor" of R is defined as

$$(1) \qquad z(R) = p - q$$

Lemma (zero favor of XOR): Let R1 and R2 be two independent random variables taking the values of 0 and 1. The zero favor of the XOR of R1 and R2 is the product of the zero favors of R1 and R2.

Proof: Let R be the XOR of R1 and R2. Using the obvious notation it follows from independence that

$$(2)\ p = p1p2 + q1q2$$

$$(3)\ q = p1q2 + q1p2$$

so it follows that

$$(4)\ z(R) = p1(p2-q2) + q1(q2-p2) = (p1-q1)(p2-q2) = z(R1)z(R2)$$

which completes the proof.

Theorem (Error Detection): (A) Suppose T is a set of faulty channels, so that T' = all channels not in T = a set of error-free channels. Any parity chunk on a channel of T' whose row in the Wiencko matrix has a zero intersection with all channels of T will show no discrepancy when compared with the parity computed from the data in its stripe set. (B) Let the assumptions be as in (A) and, further, suppose that for a set of k full related sets, all delta data and parity bits from channels of T are independently random (a burst error) such that each bit has a zero favor of absolute value less than or equal to $u<1$.

Then the probability of a nonzero error bit at any of the parity locations in any of the k related sets is always greater than or equal to

$$(1 - u^{r+s})/2$$

where r is the number of ones in the intersection of its row of the Wiencko matrix with all the channels of T, and s is 1 if the parity is in T and 0 if the parity is not in T.

Proof: (A) is a trivial consequence of the encoding scheme as described earlier above. (B) follows because the error bit is the XOR of r+s bits, each with a zero favor between u and -u. (Data or parity bits that are not in T do not contribute to the error bit, nor do data bits that meet a 0 in this parity bit's row of the Wiencko matrix.) Applying the Lemma to the XOR of these r+s random variables gives

$$-u^{r+s} <= z(\text{error bit}) <= u^{r+s}$$

from which the claim follows. This completes the proof.

As a consequence of the theorem, it is extremely probable for a large burst size that all the parity chunks not included in (A) of the theorem will, in fact, exhibit errors. Thus, examining which parity chunks exhibit errors, in conjunction with the pattern of zeroes in the Wiencko matrix, yields high probability information on which channels are good and which exhibit errors.

The error detection technique depends on the Wiencko matrix. For each set T of faulty channels there is a unique maximal set R(T) of parity chunks that satisfy the hypotheses of (A) of the theorem. R(T) may be empty. Clearly if T is a subset of U then R(U) is a subset of R(T). Also if T is empty then R(T) is all the parity chunks, where if T is nonempty then R(T) is not all the parity chunks (it must exclude, for instance, the parities in channels of T). A parity row "hits" a channel if either that parity chunk is a part of that channel, or that row in the Wiencko matrix has a nonzero intersection with that channel. R(T) is the set of parities that do not hit any member of T. A Wiencko matrix is "favorable to mc", for 0<=mc<=m, if every T of mc faulty channels has the property that R(T) is strictly greater than R(Tp) for any Tp that strictly includes T. That is, for each channel not in T, at least one parity in R(T) hits that channel. If there are no parity errors in R(T) during a long burst, it can be concluded that the bad channels are a subset of T. The reason is as follows: Suppose the one of the channels not in T is bad. That channel hits at least one of the parities in R(T) and therefore, by (B), the chances of an error on that parity over the long burst are very high. This error however is a parity error in R(T).

A simple example below is a Wiencko array of type (5, 2) with H=3 and Q=2, which when transposed becomes a Wiencko array of type (5, 3) with H=2 and Q=3:

```
                                     0  1  2  3  4 - hits(5,2)

0 0 0|0 0 0|0 1 0|1 0 0|0 1 0        x     x  x  x
     |     |     |     |
0 0 0|0 0 0|0 0 1|0 0 1|1 0 0        x     x  x  x
-----|-----|-----|-----|-----
0 1 0|0 0 0|0 0 0|0 1 0|1 0 0        x  x     x  x
     |     |     |     |
1 0 0|0 0 0|0 0 0|0 0 1|0 0 1        x  x     x  x
```

```
-----|-----|-----|-----|-----
1 0 0|0 1 0|0 0 0|0 0 0|0 1 0      x  x  x     x
     |     |     |     |
0 0 1|1 0 0|0 0 0|0 0 0|0 0 1      x  x  x     x
-----|-----|-----|-----|-----
0 1 0|1 0 0|0 1 0|0 0 0|0 0 0      x  x  x  x.
     |     |     |     |
0 0 1|0 0 1|1 0 0|0 0 0|0 0 0      x  x  x  x
-----|-----|-----|-----|-----
0 0 0|0 1 0|1 0 0|0 1 0|0 0 0         x  x  x  x
     |     |     |     |
0 0 0|0 0 1|0 0 1|1 0 0|0 0 0         x  x  x  x

x x x         x x x   x x x     0 - hits(5,3)
x x   x x x         x x x   x   1
x   x x x   x x x         x x   2
  x x x   x x x   x x x         3
        x x x   x x x   x x x   4

          s r       s
```

In the (5, 2) case, this technique can distinguish any one-channel loss from any other. The (5,3) case is more interesting. The technique can distinguish any two-channel loss from any other in this case. For instance, marked with an r is the set R({0,1}) and with an s the set R({0,2}).

<u>Detailed Practical Application of Codes and Algorithms for Data Storage.</u>

Referring now to FIG. 14, a block diagram of a system 1400 embodying inventive aspects is shown. System 1400 comprises a network and a network-attached server (NAS) 1402, where the network is represented in part by a hub or switch 1404 connected to a plurality of hosts 1406. NAS 1402 includes a host 1408 connected to a plurality of disk drives 1412 through a bus interface 1410. Disk drives 1412 are "packaged" in a number of preferably six, twelve, or eighteen. Host 1408 may also be connected to a control terminal 1424 through a port 1426. Bus interface 1410 may utilize any suitable bus technology, for example, Peripheral Component Interconnect (PCI), Small Computer System Interface (SCSI), Fibre Channel, or Integrated Drive Electronics (IDE). In addition, control terminal 1424 may be any suitable control device, such as a VT-100 terminal, and port may be a COM1 port. Alternatively, control terminal 1424 could be used with a Telnet connection.

Host 1408 utilizes an operating system 1412, a device driver 1414, a device driver 1416, and a serial I/O driver 1418. Host 1408 also includes a redundancy data generator/data recovery component 1422, which communicates with device driver 1416 in accordance with an Application Program Interface (API) 1420. Component 1422 also communicates with bus interface 1410 for use in connection with disk drives 1412. Operating system 1412 may be any suitable operating system, such as Disk Operating System (DOS), NT, Unix, Linux, etc. Coupled to the network, device driver 1414 may be implemented with one of a number of different technologies, such as PCI, 10/100BaseT (high-speed Ethernet), Fiber Distributed Data Interface (FDDI), or Copper Distributed Data Interface (CDDI).

Component 1422 is operative with disk drives 1412 as described above. Component 1422 embodies inventive principles described above (and below), and provides system 1400 with the advantages. On one hand, component 1422 may provide a fixed amount of data redundancy in the system. Alternatively, component 1422 may provide for a selection or adjustment of data redundancy. Control terminal 1424 is an administrative terminal, utilized by one who has appropriate rights to control the particulars of system 1400. If component 1422 allows for adjustments to data redundancy, control terminal 1424 provides the interface to input such control data. The amount of redundancy to be provided may be in the form of the maximum number of drives that can be lost simultaneously, or some other value or indication. A user of one of hosts 1406 may alternatively provide control data to adjust the amount of data redundancy, given the appropriate system rights and authentication.

FIG. 15 is a schematic block diagram of a controller 1500 for a disk array. Controller 1500 includes a processor 1502, a non-volatile memory

1504, a host interface 1506, a cache 1508, and a plurality of I/O processors 1510. Non-volatile memory 1504, which may be (e.g.) a read-only memory (ROM) or electrically erasable/programmable ROM (EEPROM), has software stored therein for execution by processor 1502. The plurality of I/O processors 1510 include I/O processors 1512, 1514, and 1516, which may be (e.g.) SCSI I/O processors (SIOPs). Controller 1500 also includes circuitry which provides for redundancy data generation and data recovery as described above, and is shown as a PGA 1520. All of these components are coupled to a bus 1518.

A typical command flow for a "write" operation for controller 1500 is now described. A host issues a write command to controller 1500 to write data to a logical drive. Processor 1502 initiates a command to PGA 1520 to setup the appropriate instruction type. Processor 1502 sets up the host interface 1506 to transfer data to cache 1508. Upon completion of the transfer, host interface 1506 notifies processor 1502. Processor 1502 determines that some data may be required from SCSI disks for parity computation, and instructs the I/O processors 1510 to retrieve this data. I/O processors 1510 transfer data from the disks through PGA 1520 to cache 1508 and notify processor 1502 upon completion. Processor 1520 sends this data through PGA 1520 to thereby compute the appropriate redundancy data. Processor 1502 instructs I/O processors 1510 to transfer the data, as well as the redundancy data, to the disk drives. I/O processors 1510 transfer data from cache 1508 to the disks and notify processor 1502 upon completion. Processor 1502 in turn notifies the host.

A typical command flow for a "read" operation for controller 1500 is now described. A host issues a read command to controller 1500 to read data from a logical drive. Processor 1502 initiates a command to PGA 1520 to setup the appropriate instruction type. Processor 1502 determines that the data resides on multiple disks and instructs I/O processors 1510 to retrieve data from the disk drives to cache 1508. I/O processors 1510 transfer data from the disks through PGA 1520 to cache 1508 and notify processor 1502 upon completion. The data is transferred through PGA 1520 to either provide a "pass-through," or data recovery as described above if some disk or data is unavailable. Processor 1502 sets up host interface 1506 to transfer data from cache 1508 to system memory. Upon completion, host interface 1506 notifies processor 1502, which in turn, notifies the host.

Preferably, the data redundancy generation and the data recovery is performed using a special processor technology which allows operations to be performed simultaneously on multiple units of data. This technology is available today in the form of Multimedia Extensions (MMX) by Intel Corporation.

"Disk Loss Insurance" Adjustability. Another aspect of the invention gives users a choice of how much redundancy to have as part of a disk array. The selected redundancy is made immediately and continuously useful throughout the operational life of the disk array. It is a natural extension of traditional RAID technology, one that offers performance advantages no matter what degree of redundancy in the disk array is desired.

Referring now to FIG. 16.2, a flowchart describing a method for use in redundancy adjustability is shown. Beginning at a start block 1600, control data that is indicative of an amount of data redundancy desired in the disk array is received (step 1602). The control data is indicative of the value "m" and may take a variety of forms; for example, it may be the value m or m/n. Based on this control data, one of multiple sets of data redundancy functions (or codes) are selected (step 1604). One of multiple sets of data recovery functions (codes) is selected based on the control data (step 1606). The flowchart ends at a finish block 1608. This method may be performed once to configure the system, or may be executed each time data is stored on the disk array (FIG. 17) and each time data is read from the disk array (FIG. 18).

Referring now to FIG. 16.1, a schematic block diagram of one example of an adjustable redundancy and recovery apparatus 1610 is shown. In this example, apparatus 1610 is operative in connection with five disks (n=5) and m is adjustable. Apparatus 1610 includes a plurality of function set components 1612 coupled to selectors 1622 and 1624. In this example, there are four function set components, namely function set components 1614, 1616, 1618, and 1620 associated with m=1, m=2, m=3, and m=4 respectively. Selectors 1622 and 1624 are operative to select one of function set components 1612 for redundancy data generation and data recovery. Redundancy control signals are fed to selectors 1622 and 1624 to establish which one of function set components 1612 are selected. Apparatus 1610 may be implemented in many suitable ways, such as with software or hardware as described in relation to FIGs. 6 and 7.

<u>Conventional RAID Comparisons.</u> RAID-1, "disk mirroring," allows for two disks worth of information to be copied to two other disks. RAID-1 fails, though, if the "wrong" two of the four disks fail. A "4/2" configuration allows any two of the four disks to fail while still allowing the original two disks of information to be fully recovered. Because RAID-1 solutions are costly, many databases rely strictly upon RAID-5 with striping and parity for protection against drive failure. However, RAID-5 supports continued operation only in the event of a single inoperable drive at any one moment. Losing two or more drives under RAID-5 brings operations quickly to a halt. For the cost of adding just one more drive, storage system 100 mitigates the risk of data loss by providing the means to sustain up two drive failures.

One of the better conventional protections for critical information today is accomplished through RAID-1 (mirroring), overlaying RAID-5 (striping with parity) and then adding a global hot spare. For example, if data consumes four disk drives, then reliability can be improved by replicating this data on a second "stack" of four drives. Within each stack, however, losing just one drive would make the whole database useless. To enhance reliability, each mirrored stack can be configured as an individual RAID-5 system. Since using parity adds the need for an additional drive, user data and parity information are now striped across five drives within each stack. This provides protection against the loss of a single drive within each stack. So, from an original database that required just four drives, this RAID configuration has grown to include: four drives for the original data; four drives for the mirrored data; one parity-drive (equivalent) for each stack (two in total); and one global hot spare (standby drive on which data can be rebuilt if a drive fails).

Referencing back to FIG. 2, a conventional RAID architecture shown requires a total of eleven disk drives: Here, seven drives have been added to protect data on the four original drives. The configuration can recover from a failed drive in either stack. Even if all the drives in one stack failed, the remaining drives in the surviving stack would still provide access to critical data. However, in this case, only one drive failure in the remaining stack could be tolerated. If multiple drive failures occur within each stack, the data cannot be recovered.

For comparison, an exemplary configuration based on principles of the invention is shown in FIG. 3, where equal coverage against multiple drive failure is achieved. This configuration provides protection against two-drive failure, but at a much less cost and with superior performance: requires only 7 disk drives compared to 11 for traditional RAID; doesn't tax system resources by requiring double "writes"; has faster transfer rates; requires less administrative overhead; if these disk drives cost $1,000 each, for example, the inventive apparatus saves $4,000 while providing better insurance, since any two random drives can fail and the system will continue to properly function.

The "m=1" case of the inventive techniques may be viewed as a functional equivalent of a RAID-5 array with an efficiency advantage. With the inventive techniques, both data and coding are placed on all of the disks. When no disks have failed, the inventive techniques may make slightly better use of the disk array for data reads because all n spindles of the array are utilized to retrieve clear data with no need for decoding.

An additional performance advantage of the inventive techniques is realized by converting passive "hot spares" into active contributors to the disk array. All running disks can be fully employed, which means complete utilization of all spindles for better writing and reading efficiency. This also prevents the problems of hot spares possessing latent defects, defects that might not become known until the hot spare is actively used in the array.

In a traditional RAID-1 (or 0+1, 5+1, etc.) storage configuration, with data mirrored on two independent SCSI channels, all data could be lost in one channel and operation would continue. However, if more than one drive failure concurrently occurs in both mirrored channels, then the entire storage system becomes inoperable. With a storage system according to the invention, on the other hand, multiple drive failures are sustainable.

As described, the inventive aspects are based on a pattern with the matrix designation n.m, where n>m>0, and n is the total disk count and m is the number of disks that can be lost. According to one aspect of the invention, any combination of up to m disks can be lost and the amount of data stored is equal to n-m, which is the theoretical maximum when m disks are lost. The read speed is n and the write speed is n-m, which are also theoretical maxima. Using some of these performance values, the following table compares the prior art with some inventive embodiments.

	A	H	H	B	H	H*	C	H	H
Case	0+1	10.2	10.5	5+1	10.3	10.6	5+5	9.3	9.5
Safe Loss Count	1	2	5	3	3	6	3	3	5

Max Loss Count	5	2	5	6	3	6	5	3	5
Data Capacity	5	8	5	4	7	4	4	6	4
Read Speed	#5	10	10	#5	10	10	#3	9	9
Write Speed	5	8	5	4	7	4	4	6	4

These speeds can be improved on certain specialized operating systems.

Other Related Methods, Apparatus, and Applications. FIG. 19 is a block diagram of a communication system 1900. In this embodiment, communication system 1900 is a radio frequency (RF) communication system providing for wireless communications. Communication system 1900 includes a communication device 1902 having an antenna 1904, and a communication device 1906 having an antenna 1906. Communication devices 1902 and 1906 can be portable or non-portable electronic devices, such as portable telephones, personal digital assistants (PDAs), computers (desktop or laptop), fixed base stations, etc.

As shown in FIG. 20, communication device 1902 is shown to have an RF transceiver 2002 (an RF receiver and transmitter) coupled to antenna 1904, a memory 2004, a processor 2006 (or processors), and an input/output (I/O) device(s) 2008. Processor 2006 may be or include one or more microcontrollers, microprocessors, digital signal processors, etc. I/O device 2008 may be or include memory for storing and retrieving data, a speaker and/or microphone for audio communication, a display for text or video data, etc.

Generally, communication device 1902 embodies similar features as described above in relation to Wiencko coding and operates accordingly, and otherwise operates in a conventional manner using conventional components. The inventive components of communication device 1902 may include software and/or hardware to implement the redundancy generation and recovery functions as described above. For example, communication device 902 may include a PGA as described above, or software stored in memory 2004 for execution by processor(s) 2006. The variables m and n may be any suitable numbers, and may remain fixed during operation.

FIG. 21 is a representation of a data format in communication system 1900 of FIG. 19. Variables m, n, H, Q, and D are represented and are defined similarly as above. A plurality of channels are shown, channels 1, 2, and 3 or channels 2102, 2104, and 2106, respectively. The channels may be formed using frequency division, time division, code division, etc., techniques. Since there are three channels in this example, n=3. Here, data from all three channels combine to form the user data desired. For example, a grouping 2108 from each of channels 2102, 2104, and 2106 form what would be a stripe set in a disk array. Similarly, a grouping 2110 forms what would be the next stripe set in the disk array.

FIG. 22 is the same data format as shown in FIG. 21, but where it is established that m=2 (as an example) and that particular data groupings are unavailable (due to, e.g., data errors). The unavailability is represented by an "X" over the data that are unavailable. In accordance with the present invention, user data are recoverable when any combination of two channel failures occur (m=2). For data grouping 2108 in FIG. 22, the user data is recoverable since only a single "D" from channel two is unavailable. For the next data grouping in FIG. 22, data grouping 2110, the user data are recoverable as well since only two "D"s are unavailable (data from channels two and three). For the next data grouping in FIG. 22, the user data are partially recoverable since three "D"s from all three channels are unavailable (> m), where some data are "clear" bits. Finally, for the next data grouping in FIG. 22, the user data are recoverable since only two "D"s are unavailable (data from channels one and two). Good performance preferably involves a system in which the channel demarcations are chosen such that errors tend to concentrate in a few channels rather than be spread across many channels.

FIG. 23 is another example of a data format which may be utilized in connection with communication system 1900 of FIG. 19. As an example to illustrate this format, a data grouping 2302 represents what would otherwise be a stripe set in a disk array. Similarly, a data grouping 2304 represents what would otherwise be the next stripe set in the array. Referring to FIG. 24, where m=2 and n=3, the user data is recoverable in both situations shown since no more than two "time slots" in each data grouping are unavailable.

FIG. 24.1 is a flowchart describing a method of processing data in a communication device and system, such as that described in relation to FIGs. 19-24. At one or more remote stations, RF signals are modulated with the data and transmission is provided over a plurality of n channels. Beginning at a start block 2400, the modulated RF signals are received and demodulated (step 2401). The data is received in n data groupings (step 2402), some or all of which may be adversely affected with errors. For each data grouping, errors are detected and corrected to the extent possible using conventional error detection and correction techniques (e.g., with suitable conventional error detection and correction codes). Some data groupings, however, may have errors that are not correctable with use of such conventional techniques.

If data recovery is then necessary (step 2406), the data is recovered using a data recovery function(s) (step 2408). The appropriate data recovery function(s) is selected based on which data groupings are in error, even after application of conventional techniques. If no data recovery is necessary at step 2406, or after data recovery is performed at step 2408, the data is processed (step 2410). Processing the data in step 2410 may involve processing a predetermined function based on the data (e.g., for command or control data) or processing the data for use with an I/O device (e.g., for audio, text, or video data). Other suitable error detection techniques may be used in relation to steps 2404 and 2406 (e.g., low signal strength indications, low bit error rates (BER), Wiencko code error detection, etc.).

Communication system 1900 may also operate to provide data redundancy adjustments similar to that provided and described in relation to disk arrays. Redundancy adjustability may be provided -- even dynamically during device operations. For example, communication system 1900 operates such that more redundancy is provided (i.e., m is incremented) in response to the detection of inadequate communication, and less redundancy is provided (i.e., m is decremented) otherwise. Also preferably, more redundancy is provided in response to less system capacity made available (due to, e.g., other users), and less redundancy is provided in response to more system capacity made available. The control of such adjustments may be made by the base or mobile station, or both.

Other inventive aspects described herein involve local area networks (LANs). FIG. 25 is a block diagram illustrating such an environment. A LAN 2500 includes a department 2504 connected to a switch 2506, a department 2508 connected to a switch 2510, and a server 2512. A switch controller 2502 has a number of ports which are connected to switches 2506, 2510 and server 2512. More particularly here, switch controller has twelve ports, the first four of which are connected to switch 2506, the second four of which are connected to switch 2510, and the last four of which are connected to server 2512.

Generally, the LAN and the entire system in FIG. 25 embody similar features as described above in relation to Wiencko coding and operate accordingly, but otherwise operate in a conventional manner using conventional components. The inventive components may include software and/or hardware to implement the redundancy and recovery functions, as described above. For example, switches 2506, 2510 and server 2512, as well as switch controller 2502, may include a PGA as described above, or software stored in memory for execution by a processor. As shown in FIG. 25, the system is configured for the situation where m=2 and n=4. The variables m and n may be any suitable numbers, and may remain fixed during operation. Redundancy adjustments may be provided as well, similar to that described above.

Other inventive aspects described herein involve wide area networks (WANs). FIG. 26 is a block diagram illustrating such an environment. As shown, a WAN 2600 includes a switch controller 2602 connected to one or more telephone companies (as shown, AT&T, MCI, and Sprint). More particularly, switch controller 2602 has twelve ports, the first four of which are connected to AT&T, the second four of which are connected to MCI, and the last four of which are connected to Sprint.

Generally, the WAN and the entire system in FIG. 26 embodies similar features described above in relation to Wiencko coding and operates accordingly, and otherwise operates in a conventional manner using conventional components. The inventive components may include software and/or hardware to implement the redundancy and recovery functions, as described above. For example, switch controller 2602 and the service companies, may utilize a PGA as described above, or software stored in memory for execution by a processor. As shown in FIG. 25, the system is configured for the situation where m=8 and n=12. The variables m and n may be any suitable numbers, and may remain fixed during operation. Redundancy adjustments may be provided as well, similar to that described above.

Other inventive aspects described herein involve multiple servers, or multiple servers and the Internet. FIG. 27 is a diagram illustrating such an environment. A system 2710 includes a plurality of servers 2712, such as servers 2702, 2704, and 2706. Servers 2712 may be connected via the Internet 2700. An end user 2708 may connect to the Internet 2700 to access data from one or more of servers 2712.

Generally, system 2710 embodies similar features as described above in relation to Wiencko coding and operates accordingly, and otherwise operates in a conventional manner using conventional components. The data is spread over multiple servers in a manner similar to that described in relation to a disk array. The inventive components may include software and/or hardware to implement the redundancy and recovery functions, as described above. For example, servers 2712 and user 2708 may operate in connection with a PGA as described above, or software stored in memory for execution by a processor. The variables m and n may be any suitable numbers, and may remain fixed during operation. Redundancy adjustments may be provided as well, similar to that described above.

In an alternate embodiment, system 2710 may operate using a "temporal proximity" technique. The data is spread over the multiple servers in a manner similar to that described in relation to a disk array. To illustrate, suppose m=2 and n=3, and that servers 2712 are geographically separated by some large distance. For example, server 2702 (designated "A") is located in California , server 2704 (designated "B") is located in Chicago, Illinois, and server 2706 (designated "C") is located in New York. At different times of the day, servers 2712 are loaded differently. Therefore, when user 2708 requests data from any one of servers 2712, the time it takes to retrieve that data depends upon which server is accessed. Exemplary differences in data access time from servers 2712 are summarized in the tables of FIGs. 28, 29, and 30. As shown in these tables, data access time is best from server 2702 at 7:30 a.m., from server 2704 at 8:30 a.m., and from server 2706 at 11:30 a.m.

In operation, user 2708 sends a request for some information (e.g., a file) from each one of servers 2712. In response to the requests, servers 2712 operate in a parallel fashion to submit data from the file to user 2708. Eventually, one of servers 2702 will be the first to supply user 2708 with the data (i.e., packet or group of packets). Once user 2708 receives the data from the first "winning" server, it ignores the other data submitted by the "losing" server(s). Since m=2 and n=3, user 2708 can afford to ignore this later sent data and recover the data using the techniques described herein. This method repeats for each packet or group of packets. Thus, in this embodiment, data is always made "available" from a single server.

FIG. 31 is a flowchart describing a method of processing data in accordance with such temporal proximity techniques. Assume m=n-1. Beginning at a start block 3100, a data request is submitted to n servers (step 3102). A response is received from one of n servers that is first in time (step 3104). The data is constructed using a Weincko data recovery function and that received from the first responding server (step 3106). The appropriate data recovery function is selected based on which server first responded. Responses from the other n-1 servers that are later in time may be ignored. If it is determined that all data have been received (step 3108), the flowchart ends at a finish block 3110 (but the method may repeat for subsequent requests). If all of the data have not been received, the method repeats at step 3104. This technique may be used in connection with values of m that are less than n-1 as well, where the first (n-m) responses are used to construct the data.

In another embodiment related to FIG. 31, step 3104 receives a response that is both first-in-time and error-free (based on the use conventional error detection and/or correction techniques). Here, for example, the first-in-time response may have non-correctable errors, but the second-in-time response may have no errors and therefore be used to recover the data in its entirety.

Further Related Methods, Apparatus, and Applications. Parity in memory and low level storage is intimately connected with low-level hardware and firmware so as to be transparent and not costly from the point of view of all data using applications including drivers. It is usually accomplished on the bit-level before the data is assembled into bytes or words. Though closely related to RAID, there are some differences: RAID must be able to operate as an overlay on any disks (or other storage devices) independent of the hardware, firmware or memory structure. It may also handle data in comparatively large blocks to avoid data access inefficiency, or in small blocks to avoid latency.

File and system copying and backup has a relationship with mirroring and RAID-1 because a copy is made. However, RAID must be available at all times to recover on-the-fly from device loss. Backups are periodic in order to avoid overburdening the system: they therefore can return data that is out-of-date. But what they do return is unlimited by the nature of the failure which can be total, unlike RAID.

Similar methods and apparatus may also be utilized to create a real-time backup capability that works in a fundamentally different way than present backup capabilities. Present "tape backup" schemes are an all-or-nothing proposition, and fail to accommodate the cases where a real-time backup, but not complete backup, is what fits the application. This application extends the RAID field in the system beyond that of the homogeneous disks. As an example, consider an 18-disk disk array, along with 5 additional storage elements, say 5 tape drives. The encoding may be setup for a "23/7" configuration, so that the equivalent of 16 disks worth of information is being stored. Here, the disk array's information alone would be sufficient to read all data, including if up to two disks failed in the disk array. If up to five additional disks failed, then data is still recoverable with a little help from the tape drives.

In the exemplary case, the risk of losing all data if more than seven disks fail may be acceptable, but the backup capability may need to be updated in real-time. A downside to this approach is that decode is always necessary, because access to the slower tape drives when at least sixteen disks are available is never desired, and decode must be performed for five elements even when all eighteen disks are available. However, this downside can be ameliorated using fast-acting decode logic.

The example above applies techniques to disks and tape. However, the techniques may be applied to faster disks and slower disks, or a number of levels of faster, fast, moderate, slow, slower storage technologies, whether they be semiconductor, rotating media or streaming media. The parallel-element redundancy techniques of the present invention can be applied to systems employing a mixture of these media.

WIENCKO CODES AND FURTHER METHODS TO OBTAIN ADDITIONAL CODES.

A. Exemplary Weincko Codes. With code searching methods, encoding matrices have been found for many pairs of n and m. APPENDIX A below provides an exemplary list of currently available Wiencko codes. Since multiple instances of Wiencko codes exist for many pairs of m and n, those with the most favorable implementation characteristics (smaller data and code chunk sizes, sparseness of encoding matrix) have been selected for application.

B. Code Determining/Validation Method. Further description is provided as source code in APPENDIX B.

C. Further Methods. Techniques have been developed to identify encoding matrices for additional pairs of m and n, especially for higher values thereof, and are provided as source code in APPENDIX C.

D. Duality Methods. These are described in APPENDIX D along with source code associated therewith.

E. Wiencko Codes Where m=1 and m=(n-1). The m=1 and m=(n-1) cases are shown in the code list in APPENDIX A, although it is clear at a glance that they are all essentially the same. It can be mathematically proven that the patterns shown will always work for m=1 and m=(n-1), for any value of n>=2. The following are the proofs using n=5 cases as examples.

Example:

```
5 1 4 1
1000 -
0100 -
```

```
0010 -
0001 -
```

The general solution in the m=1 case shows an (n-1)x(n-1) identity matrix following the numbers n 1 n-1 1 thus for p=1. The actual parity matrix is expanded from this by adding a row of zeroes on the top then rotating and copying this (n-1)xn column to the right until an (n*(n-1))xn parity matrix exists. There is only one case to test - removal of the first disk (all other 1 disk removals are rotations of this). The resulting square matrix (parity rows from the last n-1 disks, data columns from the first disk) is an identity matrix, and is therefore invertible.

Example:

```
5 4 1 4
0 -
0
0
1
0 -
0
1
0
0 -
1
0
0
1 -
0
0
0
```

The general solution in the m=n-1 case consists of a column of n-1 columns, each of which has n-1 entries. The k-th little column (counting from 1) has the n-k-th value (counting from 1) equal to 1, the others 0. To get the actual parity matrix, as above, you add a little column on top and then copy to the right, rotating each time. As above, there is essentially only one test;

removing all the disks but the first. The square matrix uses data columns from the last (n-1) disks and parity rows from the first disk. The result is an identity matrix, which is always invertible.

Conclusion. As readily apparent, the various inventive aspects described herein provide several advantages in simplicity and constitutes a breakthrough and a technological leap in the arts, as in the fields of data redundancy generation and recovery, data retrieval and storage, data communications, and network operations. The present invention is a particular breakthrough in the field of RAID architecture, which has long used rather limited techniques. Thus, the scope of all of the claimed inventions should be understood to be quite broad and warrant a broad range of equivalent structures and functionalities.

APPENDIX A

List of representative Wiencko codes. Mulitple solutions may exist for a given m and n, and thus the codes below are merely preferred examples.

The first line associated with each set has four numbers; the first is n, the second is m, the third is H, and the fourth is Q (Q = D-H). It follows that p=gcd(H, Q).

```
4 1 3 1
100 -
010 -
001 -

4 2 1 1
1 -
1 -
0 -

4 3 1 3
0 -
0
1
0 -
1
0
1 -
0
0

5 1 4 1
1000 -
0100 -
0010 -
0001 -

5 2 3 2
010 -
100
100 -
001
010 -
001
000 -
```

A1

```
000

5 3 2 3
00 -
01
10
10 -
00
01
10 -
01
00
00 -
00
00

5 4 1 4
0 -
0
0
1
0 -
0
1
0
0 -
1
0
0
1 -
0
0
0

6 1 5 1
10000 -
01000 -
00100 -
00010 -
00001 -

6 2 2 1
10 -
10 -
01 -
00 -
01 -

6 3 1 1
0 -
1 -
1 -
```

A-2

```
1 -
0 -

6 4 1 2
0 -
1
0 -
1
1 -
0
0 -
0
1 -
0

6 5 1 5
0 -
0
0
0
1
0 -
0
0
1
0
0 -
0
1
0
0
0 -
1
0
0
0
1 -
0
0
0
0

7 1 6 1
100000 -
010000 -
001000 -
000100 -
000010 -
000001 -

7 2 5 2
10000 -
```

A-3

```
01000
00100 -
10000
01000 -
00001
00100 -
00010
00010 -
00001
00000 -
00000

7 3 4 3
1000 -
0100
0010
0001 -
1000
0100
1000 -
0010
0100
0000 -
0010
0001
0001 -
0000
0000
0000 -
0000
0000

7 4 3 4
000 -
100
010
001
001 -
000
100
010
000 -
010
100
001
100 -
010
000
000
001 -
000
000
```

A-4

```
000
000 -
000
000
000

7 5 2 5
00 -
00
00
10
01
00 -
00
01
00
10
10 -
00
00
01
00
00 -
10
01
00
00
10 -
01
00
00
00
00 -
00
00
00
00

7 6 1 6
0 -
0
0
0
0
1
0 -
0
0
0
1
0
0 -
```

A-5

```
0
0
1
0
0
0 -
0
1
0
0
0
0 -
1
0
0
0
0
1 -
0
0
0
0
0

8 1 7 1
1000000 -
0100000 -
0010000 -
0001000 -
0000100 -
0000010 -
0000001 -

8 2 3 1
011 -
100 -
100 -
010 -
001 -
000 -
001 -

8 3 5 3
00000 -
00001
00100
01000 -
10000
00100
10000 -
01000
00010
```

A-6

```
10000 -
00010
01000
00100 -
00010
00001
00001 -
00000
00000
00000 -
00000
00000

8 4 2 2
10 -
01
10 -
01
00 -
10
10 -
01
00 -
00
01 -
00
00 -
00

8 5 3 5
010 -
000
100
000
000
000 -
000
100
001
010
000 -
100
000
010
001
000 -
010
000
100
001
100 -
010
```

A-7

```
001
000
000
001 -
000
000
000
000
000 -
000
000
000
000


8 6 1 3
1 -
1
0
0 -
0
1
0 -
0
1
0 -
1
0
1 -
0
0
0 -
0
0
1 -
0
0

8 7 1 7
0 -
0
0
0
0
0
1
0 -
0
0
0
0
1
```

A-8

```
0
0 -
0
0
0
1
0
0
0 -
0
0
1
0
0
0
0 -
0
1
0
0
0
0
0 -
1
0
0
0
0
0
1 -
0
0
0
0
0
0

9 1 8 1
10000000 -
01000000 -
00100000 -
00010000 -
00001000 -
00000100 -
00000010 -
00000001 -

9 2 7 2
0100000 -
0001000
0010000 -
1000000
```

A-9

```
1000000 -
0100000
0010000 -
0000010
0000100 -
0001000
0000100 -
0000001
0000010 -
0000001
0000000 -
0000000

9 3 2 1
01 -
11 -
00 -
10 -
10 -
01 -
01 -
00 -

9 4 5 4
01000 -
00001
10000
00010
00000 -
00000
00001
10000
00010 -
01000
00100
10000
00010 -
10000
00100
01000
01000 -
00100
00000
00001
00100 -
00010
00001
00000
00000 -
00000
00000
00000
```

A-10

```
00000 -
00000
00000
00000

9 6 1 2
1 -
0
1 -
1
0 -
0
0 -
1
0 -
1
1 -
0
1 -
0
0 -
0

9 7 2 7
00 -
00
10
00
00
01
00
00 -
00
00
10
00
00
01
00 -
00
00
00
10
00
01
00 -
00
00
00
10
01
00
```

A-11

```
00 -
00
10
01
00
00
00
01 -
10
00
00
00
00
00
10 -
01
00
00
00
00
00
00 -
00
00
00
00
00
00

9 8 1 8
0 -
0
0
0
0
0
0
1
0 -
0
0
0
0
0
1
0
0 -
0
0
0
0
1
```

A-12

```
0
0
0 -
0
0
0
1
0
0
0
0 -
0
0
1
0
0
0
0
0 -
0
1
0
0
0
0
0
0 -
1
0
0
0
0
0
0
1 -
0
0
0
0
0
0
0

18 1 17 1
10000000000000000 -
01000000000000000 -
00100000000000000 -
00010000000000000 -
00001000000000000 -
00000100000000000 -
00000010000000000 -
00000001000000000 -
```

A-13

```
0000000100000000 -
0000000010000000 -
0000000001000000 -
0000000000100000 -
0000000000010000 -
0000000000001000 -
0000000000000100 -
0000000000000010 -
0000000000000001 -

18 2 8 1
01000000 -
00001000 -
10000000 -
00100000 -
10000000 -
00000100 -
01000000 -
00010000 -
00100000 -
00000010 -
00010000 -
00001000 -
00000100 -
00000010 -
00000001 -
00000001 -
00000000 -

18 3 5 1
00100 -
00000 -
00010 -
10000 -
00100 -
10000 -
00010 -
00001 -
00001 -
01000 -
10000 -
01000 -
01000 -
00100 -
00010 -
00001 -
00000 -

10 2 4 1
0100 -
1000 -
0010 -
```

A14

```
1000 -
0100 -
0010 -
0001 -
0001 -
0000 -

10 3 7 3
0000001 -
0000100
0100000
0000010 -
0010000
0001000
1000000 -
0010000
0000100
1000000 -
0100000
0001000
1000000 -
0000010
0100000
0010000 -
0001000
0000100
0000000 -
0000010
0000001
0000001 -
0000000
0000000
0000000 -
0000000
0000000

10 8 1 4
0 -
0
1
0
0 -
0
0
1
0 -
1
0
0
0 -
0
0
```

A-15

```
1
0 -
0
1
0
0 -
1
0
0
1 -
0
0
0
1 -
0
0
0
0 -
0
0
0

11 2 9 2
100000000 -
000001000
000010000 -
100000000
001000000 -
010000000
010000000 -
000100000
001000000 -
000000100
000100000 -
000010000
000001000 -
000000100
000000010 -
000000001
000000010 -
000000001
000000000 -
000000000

11 9 2 9
00 -
00
00
10
00
00
00
```

A-16

```
00
01
00 -
00
00
00
01
00
00
00
10
00 -
00
00
00
00
00
01
10
00
00 -
00
00
00
00
10
00
01
00
00 -
00
10
00
00
00
01
00
00
00 -
00
00
00
10
01
00
00
00
00 -
00
10
01
00
```

A-17

```
00
00
00
00
10 -
01
00
00
00
00
00
00
00
10 -
01
00
00
00
00
00
00
00
00 -
00
00
00
00
00
00
00
00

12 10 1 5
0 -
0
0
1
0
1 -
0
0
0
0
0 -
0
0
0
1
0 -
0
1
0
```

A-18

```
0
0 -
0
0
0
1
0 -
0
0
1
0
0 -
0
1
0
0
0 -
1
0
0
0
0 -
1
0
0
0
1 -
0
0
0
0
0 -
0
0
0
0

12 2 5 1
01000 -
00001 -
10000 -
00100 -
10000 -
01000 -
00100 -
00010 -
00010 -
00001 -
00000 -

12 3 3 1
010 -
```

A-19

```
001 -
100 -
100 -
001 -
100 -
010 -
010 -
000 -
001 -
000 -

12 9 1 3
0 -
1
0
1 -
0
0
0 -
0
1
0 -
0
1
1 -
0
0
0 -
0
1
0 -
1
0
0 -
1
0
0 -
0
0
1 -
0
0
0 -
0
0

13 2 11 2
00010000000 -
00100000000
10000000000 -
01000000000
00001000000 -
```

A-20

```
10000000000
00000010000 -
00000100000
01000000000 -
00100000000
00010000000 -
00001000000
00000000100 -
00000100000
00000010000 -
00000001000
00000001000 -
00000000100
00000000010 -
00000000001
00000000010 -
00000000001
00000000000 -
00000000000

14 12 1 6
0 -
0
1
0
0
0
0 -
0
0
0
0
1
0 -
0
0
0
1
0
1 -
0
0
0
0
0
0 -
0
0
0
0
1
0 -
```

A-21

```
0
0
1
0
0
0 -
0
0
0
1
0
0 -
0
0
1
0
0
0 -
0
1
0
0
0
0 -
1
0
0
0
0
0 -
1
0
0
0
0
0 -
0
0
0
0
0
1 -
0
0
0
0
0

14 2 6 1
000100 -
100000 -
010000 -
```

A-22

```
000001 -
100000 -
001000 -
010000 -
001000 -
000100 -
000010 -
000010 -
000000 -
000001 -

15 2 13 2
1000000000000 -
0100000000000
0010000000000 -
1000000000000
0001000000000 -
0000010000000
0000000100000 -
0000100000000
0100000000000 -
0000001000000
0010000000000 -
0001000000000
0000100000000 -
0000010000000
0000000010000 -
0000001000000
0000000000100 -
0000000100000
0000000010000 -
0000000001000
0000000001000 -
0000000000100
0000000000010 -
0000000000001
0000000000010 -
0000000000001
0000000000000 -
0000000000000

16 2 7 1
1000000 -
0001000 -
0010000 -
0100000 -
1000000 -
0000010 -
0100000 -
0010000 -
0001000 -
0000100 -
```

A-23

```
0000100 -
0000010 -
0000001 -
0000000 -
0000001 -

17 2 15 2
000100000000000 -
000010000000000
000000001000000 -
010000000000000
000010000000000 -
000000000010000
000000000000010 -
000001000000000
000000100000000 -
000000000001000
000000000000001 -
000000001000000
100000000000000 -
001000000000000
000000000100000 -
000000010000000
000000000001000 -
000000000100000
100000000000000 -
000000010000000
000000000000000 -
000001000000000
000000000000001 -
000000000000100
000000000000100 -
000000000000000
000000100000000 -
010000000000000
001000000000000 -
000100000000000
000000000000010 -
000000000010000

10 5 2 2
11 -
11
10 -
11
11 -
01
00 -
00
10 -
11
00 -
```

A-24

```
10
01 -
00
00 -
00
00 -
00

10 7 3 7
000 -
000
000
000
000
000
000
000 -
000
000
000
000
000
100
000 -
000
000
000
000
010
001
000 -
000
100
010
001
000
000
100 -
001
000
000
000
010
000
100 -
010
000
001
000
000
000
100 -
```

A-25

```
000
010
000
001
000
000
000 -
000
010
001
000
100
000
000 -
001
000
000
010
000
100

13 11 2 11
00 -
00
00
00
00
00
00
00
00
00
00
00 -
00
00
00
00
00
00
00
00
10
01
00 -
00
00
00
00
00
00
00
```

A-26

```
00
10
01
00 -
00
00
00
00
00
00
10
01
00
00
00 -
00
00
00
00
00
10
01
00
00
00
00 -
00
00
00
00
01
00
00
10
00
00
00 -
00
00
10
01
00
00
00
00
00
00
00 -
10
01
00
00
```

A-27

```
00
00
00
00
00
00
00 -
00
00
00
00
01
10
00
00
00
00
01 -
00
00
00
10
00
00
00
00
00
00
00
10 -
01
00
00
00
00
00
00
00
00
00
00 -
00
01
10
00
00
00
00
00
00
00

15 13 2 13
```

A-28

```
00 -
00
00
00
00
00
00
00
00
00
00
00
00
00 -
00
00
00
00
00
00
00
00
00
00
10
01
00 -
00
00
00
00
00
00
00
00
00
00
10
01
00 -
00
00
00
00
00
00
00
00
10
01
00
00
```

A-29

```
00 -
00
00
00
00
00
00
00
10
01
00
00
00
00 -
00
00
00
00
00
00
01
00
00
10
00
00
00 -
00
00
00
00
00
01
00
10
00
00
00
00
00 -
00
00
00
10
01
00
00
00
00
00
00
00
```

A-30

```
00 -
00
10
01
00
00
00
00
00
00
00
00
00
00 -
10
00
00
00
00
01
00
00
00
00
00
00
00 -
00
00
00
01
00
00
10
00
00
00
00
00
00 -
00
00
10
00
01
00
00
00
00
00
00
00
```

A-31

```
01 -
00
10
00
00
00
00
00
00
00
00
00
00
10 -
01
00
00
00
00
00
00
00
00
00
00
00

16 14 1 7
0 -
0
0
0
0
0
1
0 -
0
0
0
0
0
0
0 -
0
0
0
0
0
1
0 -
0
0
```

A-32

```
0
0
1
0
0 -
0
0
0
1
0
0
0 -
0
0
0
1
0
0
0 -
0
0
1
0
0
0
0 -
0
1
0
0
0
0
0 -
1
0
0
0
0
0
0 -
0
0
0
0
1
0
1 -
0
0
0
0
0
```

A-33

```
0
0 -
1
0
0
0
0
0
0 -
0
1
0
0
0
0
0 -
0
0
1
0
0
0
1 -
0
0
0
0
0
0

17 15 2 15
00 -
00
00
00
00
00
00
00
00
00
01
00
00
10
00
00 -
00
10
01
00
00
```

A-34

```
00
00
00
00
00
00
00
00
00
00 -
01
00
00
00
00
10
00
00
00
00
00
00
00
00
00 -
00
00
00
00
00
00
00
00
00
00
00
10
00
00
00 -
00
00
00
00
00
00
00
00
00
00
00
01
```

A-35

```
00
10
00 -
00
00
00
00
01
00
00
00
00
00
00
00
00
00
10 -
00
00
00
00
00
00
01
00
00
00
00
00
00
00
00 -
00
00
00
00
00
00
00
00
01
00
10
00
00
00
00 -
00
00
00
00
```

A-36

```
00
00
01
00
10
00
00
00
00
00
10 -
00
01
00
00
00
00
00
00
00
00
00
00
00
00
00 -
00
00
00
00
00
00
00
01
00
00
00
00
00
10
00 -
00
00
00
00
00
10
00
00
00
00
01
```

A-37

```
00
00
00
00 -
00
00
00
00
01
00
00
00
00
00
00
00
10
00
00 -
00
00
00
10
00
00
00
00
00
01
00
00
00
00
00 -
01
00
00
00
00
00
00
10
00
00
00
00
00
00
00 -
00
00
10
```

A-38

```
01
00
00
00
00
00
00
00
00
00
00

18 15 1 5
0 -
0
0
0
0
0 -
0
0
0
1
0 -
0
0
1
0
0 -
0
1
0
0
0 -
1
0
0
0
0 -
1
0
0
0
1 -
0
0
0
0
0 -
1
0
0

-
```

A-39

```
0
0 -
0
0
0
1
0 -
0
0
0
1
0 -
0
0
1
0
1 -
0
0
0
0
0 -
0
1
0
0
1 -
0
0
0
0
0 -
0
0
1
0
0 -
0
0
0
0
0 -
0
1
0
0

9 5 4 5
0000 -
0000
0000
0000
```

A-40

```
0000
0000 -
0000
0000
0000
0000
0000 -
0000
1000
0100
0010
0000 -
1000
0100
0000
0001
0100 -
0001
0010
1000
0000
0001 -
0100
0010
1000
0000
0001 -
0000
0000
0000
0010
0010 -
1000
0000
0001
0100

18 16 1 8
0 -
0
0
0
0
0
0
0
0 -
0
0
0
0
0
```

A-41

```
0
1
0 -
0
0
0
0
0
0
1
0 -
0
0
0
0
0
1
0
0 -
0
0
0
0
1
0
0
0 -
0
0
0
1
0
0
0
0 -
0
0
1
0
0
0
0
0 -
0
0
0
0
0
1
0
0 -
0
```

A-42

```
1
0
0
0
0
0
0 -
0
0
1
0
0
0
0
0 -
1
0
0
0
0
0
0
0 -
0
0
0
0
1
0
0
1 -
0
0
0
0
0
0
0
0 -
0
1
0
0
0
0
0
1 -
0
0
0
0
0
```

A-43

```
0
0
0 -
0
0
0
1
0
0
0
0 -
1
0
0
0
0
0
0

9 5 4 5
0000 -
0000
0000
0000
0000
0000 -
0000
0000
0000
0000
0000 -
0000
1000
0100
0010
0000 -
1000
0100
0000
0001
0100 -
0001
0010
1000
0000
0001 -
0100
0010
1000
0000
0001 -
0000
```

A-44

```
0000
0000
0010
0010 -
1000
0000
0001
0100
```

A-45

```
/*{{{  includes*/
/* #include <sys/types.h>
   #include <sys/stat.h>
   #include <fcntl.h>
   #include <termios.h> */
#include <stdio.h>
#include <stdlib.h>
/*}}}  */
/*{{{  defines*/

#define DEBUG 1
#define MAX_JDIM 256
#define MAX_IDIM 306
#define MAX_LENGTH 8192
#define MAX_COUNT  1024
#define FILENAME "file.joe"
/*}}}  */
main(int argc, char *argv[])
{
  FILE *stream;
  char textline[512],filename[128],rotated[10];
  char joefile[MAX_JDIM][MAX_IDIM];
  char jumps[400][18];
  unsigned long tests[32], testmatrix[32], testm2[32], testm3[32];
  unsigned long long ltests[64], ltestmatrix[64], ltestm2[64], ltestm3[64];
  int n,m,D,P,datalines,i,j,x,y,z,ndeads,alive,ndim,mtemp,mflip;
  /*{{{  steptobit and bittostep*/
  /*
   * Use global n.
   * bitmatrix bits has 1 in position k if k-th disk is dead.
   * char steps[i] is a step from a bit to a next bit.
   * bits = steptobit(steps)
   * nbits = bittostep(steps, bits)
   * invert each other. Rotations are treated as equivalent.
   */
  int steptobit(const char steps[]) {
    int test, ind, res;
    test = 1;
    res = 0;
    ind = 0;
    while (steps[ind]) {
      res |= test;
      test <<= ((int)steps[ind]);
      ind++;
    }
    return res;
  }
  int bittostep(int bits, char steps[]) {
    int test, ind, shift, shiftlast, shiftfirst;
    test = 1;
    ind = 0;
    shiftlast = n;
    shiftfirst = -1;
    for (shift=0;shift<n;shift++) {
      if (bits&test) {
        if (shiftlast<shift) {
          steps[ind] = (char)(shift-shiftlast);
```

B-1

```
          ind++;
        } else {
          shiftfirst = shift;
        }
        shiftlast = shift;
      }
      test <<= 1;
    }
    if (shiftfirst >= 0) {
      steps[ind] = (char)(shiftfirst + n - shiftlast);
      ind++;
    }
    steps[ind] = '\0';
    return ind;
  }
  /*}}}  */
  /*{{{  void showmatrix(comment, matrix) - use ndim for dimension.*/
  void showmatrix(char comment[], unsigned long matrix[]) {
    int j,i;
    char matints[33];
    printf("%s:\n",comment);
    matints[ndim] = '\0';
    for (j=0;j<ndim;j++) {
      unsigned long murk = matrix[j];
      unsigned long imask = 1;
      for (i=0;i<ndim;i++) {
        if (murk&imask) {matints[i] = '|';
        } else {matints[i] = 'O';}
        imask <<= 1;
      }
      printf(" %s\n",matints);
    }
  }
  /*}}}  */
  /*{{{  void showmatrix64(comment, matrix) - use ndim for dimension.*/
  void showmatrix64(char comment[], unsigned long long matrix[]) {
    int j,i;
    char matints[65];
    printf("%s:\n",comment);
    matints[ndim] = '\0';
    for (j=0;j<ndim;j++) {
      unsigned long long murk = matrix[j];
      unsigned long long imask = 1;
      for (i=0;i<ndim;i++) {
        if ((murk&imask)!=(long long)0) {matints[i] = '|';
        } else {matints[i] = 'O';}
        imask <<= 1;
      }
      printf(" %s\n",matints);
    }
  }
  /*}}}  */
  /*{{{  int mul() multiplies f1 by f2 to get p saving i (bits) dim of f2*/
  int mul(unsigned long f1[], unsigned long f2[], unsigned long p[],
    int downf1, int jdim) {
    int k,j;
    for (k=0;k<downf1;k++) {
```

B-2

```
    unsigned long nurp = f1[k];
    unsigned long temp = 0;
    for (j=0;j<jdim;j++) {
      if (nurp&tests[j]) temp ^= f2[j];
    }
    p[k] = temp;
  }
}
/*}}}  */
/*{{{  int mul64() multiplies lf1 by lf2 to get lp saving bits dim of lf2*/
int mul64(unsigned long long f1[], unsigned long long f2[],
  unsigned long long p[], int downf1, int jdim) {
  int k,j;
  for (k=0;k<downf1;k++) {
    unsigned long long nurp = f1[k];
    unsigned long long temp = 0;
    for (j=0;j<jdim;j++) {
      if (nurp&ltests[j]) temp ^= f2[j];
    }
    p[k] = temp;
  }
}
/*}}}  */
/*{{{  int det() tests testmatrix to ndim<=32*/
int det(void) {
  int i, j, k, ihit;
  for (j=0;j<ndim;j++) {
    ihit = -1; /* no hit yet */
    for (i=j;i<ndim;i++) {
      if (testmatrix[i]&tests[j]) {
        if (ihit<0) {
          ihit = i;
        } else {
          testmatrix[i] ^= testmatrix[ihit];
        }
      }
    }
    if (ihit<0) return 0;
    if (ihit>j) {
      unsigned long temp;
      temp = testmatrix[ihit];
      testmatrix[ihit] = testmatrix[j];
      testmatrix[j] = temp;
    }
  }
  return 1;
}
/*}}}  */
/*{{{  int det64() tests ltestmatrix to ndim<=64*/
int det64(void) {
  int i, j, k, ihit;
  for (j=0;j<ndim;j++) {
    ihit = -1; /* no hit yet */
    for (i=j;i<ndim;i++) {
      if (ltestmatrix[i]&ltests[j]) {
        if (ihit<0) {
          ihit = i;
```

B-3

```
          } else {
            ltestmatrix[i] ^= ltestmatrix[ihit];
          }
        }
      }
      if (ihit<0) return 0;
      if (ihit>j) {
        unsigned long long temp;
        temp = ltestmatrix[ihit];
        ltestmatrix[ihit] = ltestmatrix[j];
        ltestmatrix[j] = temp;
      }
    }
    return 1;
  }
  /*}}}  */
  /*{{{  int inv() inverts testmatrix to testm2 then checks; ndim<=32*/
  int inv(void) {
    int i, j, k, ihit;
    /* create a unit matrix */
    memcpy((void *)testm2, (void *)tests, 128);
    #if DEBUG
    showmatrix("Before inversion",testmatrix);
    #endif
    memcpy((void *)testm3, (void *)testmatrix, 128);
    for (j=0;j<ndim;j++) {
      ihit = -1; /* no hit yet */
      for (i=j;i<ndim;i++) {
        if (testmatrix[i]&tests[j]) {
          if (ihit<0) {
            ihit = i;
          } else {
            testmatrix[i] ^= testmatrix[ihit];
            testm2[i] ^= testm2[ihit];
          }
        }
      }
      if (ihit<0) return 0;
      if (ihit>j) {
        unsigned long temp;
        temp = testmatrix[ihit];
        testmatrix[ihit] = testmatrix[j];
        testmatrix[j] = temp;
        temp = testm2[ihit];
        testm2[ihit] = testm2[j];
        testm2[j] = temp;
      }
      for (i=0;i<j;i++) {
        if (testmatrix[i]&tests[j]) {
          testmatrix[i] ^= testmatrix[j];
          testm2[i] ^= testm2[j];
        }
      }
    }
    memcpy((void *)testmatrix, (void *)testm3, 128);
    #if DEBUG
    showmatrix("After inversion",testm2);
```

B-4

```
    #endif
    /* now matmult testm2 by testmatrix and see that it is unitmatrix */
    mul(testm2,testmatrix,testm3,ndim,ndim);
    #if DEBUG
    showmatrix("Product",testm3);
    #endif
    for (j=0;j<ndim;j++) {
      if (testm3[j]!=tests[j]) return 0;
    }
    return 1;
  }
  /*}}}  */
  /*{{{  int inv64() inverts ltestmatrix to ltestm2 then checks; ndim<=64*/
  int inv64(void) {
    int i, j, k, ihit;
    /* create a unit matrix */
    memcpy((void *)ltestm2, (void *)ltests, 512);
    #if DEBUG
    showmatrix64("Before inversion",ltestmatrix);
    #endif
    memcpy((void *)ltestm3, (void *)ltestmatrix, 512);
    for (j=0;j<ndim;j++) {
      ihit = -1; /* no hit yet */
      for (i=j;i<ndim;i++) {
        if (ltestmatrix[i]&ltests[j]) {
          if (ihit<0) {
            ihit = i;
          } else {
            ltestmatrix[i] ^= ltestmatrix[ihit];
            ltestm2[i] ^= ltestm2[ihit];
          }
        }
      }
      if (ihit<0) return 0;
      if (ihit>j) {
        unsigned long long temp;
        temp = ltestmatrix[ihit];
        ltestmatrix[ihit] = ltestmatrix[j];
        ltestmatrix[j] = temp;
        temp = ltestm2[ihit];
        ltestm2[ihit] = ltestm2[j];
        ltestm2[j] = temp;
      }
      for (i=0;i<j;i++) {
        if (ltestmatrix[i]&ltests[j]) {
          ltestmatrix[i] ^= ltestmatrix[j];
          ltestm2[i] ^= ltestm2[j];
        }
      }
    }
    memcpy((void *)ltestmatrix, (void *)ltestm3, 512);
    #if DEBUG
    showmatrix64("After inversion",ltestm2);
    #endif
    /* now matmult ltestm2 by ltestmatrix and see that it is unitmatrix */
    mul64(ltestm2,ltestmatrix,ltestm3,ndim,ndim);
    #if DEBUG
```

B-5

```
  showmatrix64("Product",ltestm3);
  #endif
  for (j=0;j<ndim;j++) {
    if (ltestm3[j]!=ltests[j]) return 0;
  }
  return 1;
}
/*}}}  */
/*{{{  void testem() tests various matrices*/
void testem(void) {
  do {
    printf("Enter test matrix in hex X X X X:");
    scanf("%1X %1X %1X %1X",&testmatrix[0],&testmatrix[1],
      &testmatrix[2],&testmatrix[3]);
    ndim = 4;
    printf ("%1X %1X %1X %1X: det was %d\n",
      testmatrix[0],testmatrix[1],testmatrix[2],testmatrix[3],det());
  } while (testmatrix[0]);
}
/*}}}  */
/* START OF THE STATEMENTS */
if (argc>1) {
  strcpy(filename,argv[1]);
} else {
  strcpy(filename,FILENAME);
}
for(i=0;i<32;i++) {
  tests[i] = 1<<i;
}
for(i=0;i<64;i++) {
  ltests[i] = ((long long)1)<<i;
  ltestmatrix[i] = ((long long)(-1))<<i;
}
ndim = 0;
printf ("\n\nTEST PROGRAM validate READS %s...\n\n\n", filename);
alive = 1;
/*{{{  build the D to P matrix*/

stream = fopen( filename, "r");
if (stream==NULL) {
  printf ("%s does not exist.\n",filename);
  exit(1);
}
printf ("%s exists. Reading in its data.\n",filename);
if (fgets(textline,511,stream)!=NULL) {
  sscanf(textline,"%d %d %d %d",&n,&m,&D,&P);
  printf("n=%d,m=%d,D=%d,P=%d\n",n,m,D,P);
  if (n&&m&&D&&P&&(n>m)&&((n*P)==((D+P)*m))) {
    printf("Confirmed correct ratio.\n");
  } else {printf("Error in ratio or zero entry.\n");exit(1);}
}
mtemp = n-m;
if ((n>18)||((m>5)&&(mtemp>5))||((n*P)>MAX_JDIM)||((n*D)>MAX_IDIM)) {
  printf("Horsefeathers! Too big for our buffers! ixj=%dx%d\n",n*D,n*P);
  if ((n>18)||((m>5)&&(mtemp>5))) exit(1);
  alive = 0;
}
```

B-6

```
if (alive) {
  /*{{{  read in and check the data*/
  memset(joefile, (int)'0', MAX_JDIM*MAX_IDIM);
  datalines = 0;
  memset(textline,(int)'7',D);
  while (fgets(textline,511,stream)!=NULL) {
    printf(">>>%s",textline);
    for (x=0;x<D;x++) {
      if ((((int)textline[x])-48)>>1) {
        printf("Illegal character in data line %d.\n",datalines);
        alive = 0;
      }
    }
    memcpy(&joefile[datalines+P][0],textline,D);
    memset(textline,(int)'7',D);
    datalines++;
  }
  printf("Total of %d lines of data.",datalines);
  if (datalines==((n-1)*P)) {printf(" Correct.\n");
  } else {printf(" Should have been %d.\n",(n-1)*P);alive = 0;}
  /*}}}  */
}
fclose(stream);
if (alive) {
  /*{{{  fill up matrix and print it.*/
  for (i=D;i<(n*D);i+=D) {
    x = i - D;
    for(j=P;j<(n*P);j++) memcpy(&joefile[j][i],&joefile[j-P][x],D);
    y = (n-1)*P;
    for(j=0;j<P;j++) {
      memcpy(&joefile[j][i],&joefile[y][x],D);
      y++;
    }
  }
  textline[n*D] = '\0';
  for (j=0;j<(n*P);j++) {
    memcpy(textline,&joefile[j][0],n*D);
    printf("%s\n",textline);
  }
  /*}}}  */
}
/*}}}  */
/*{{{  find all possible essentially different cases of deads*/
mflip = 0;
if ((m>5)&&(mtemp<6)) {
  mflip = m;
  m = mtemp;
  mtemp = mflip;
}
if (m==1) {
  /*{{{  just 1*/
  ndeads = 1;
  jumps[0][0] = (char) n;
  jumps[0][1] = '\0';
  /*}}}  */
} else if (m==2) {
  /*{{{  n/2 of them*/
```

B-7

```
    ndeads = 0;
    for (i=1;i<=(n/2);i++) {
      jumps[ndeads][0] = (char) i;
      jumps[ndeads][1] = (char) (n-i);
      jumps[ndeads][2] = '\0';
      ndeads++;
    }
    /*}}}  */
  } else if (m==3) {
    /*{{{  special case if n divisible by 3*/
    ndeads = 0;
    textline[3] = '\0';
    for (j=1;j<=(n/3);j++) {
      for (i=j;i<(n-j-j);i++) {
        textline[0] = (char) j;
        textline[1] = (char) i;
        textline[2] = (char) (n-i-j); /* strictly bigger than j */
        strcpy(&jumps[ndeads][0],textline);
        ndeads++;
      }
      if ((3*j)==n) {
        memset(textline,j,3);
        strcpy(&jumps[ndeads][0],textline);
        ndeads++;
      }
    }
    /*}}}  */
  } else if (m==4) {
    /*{{{  m = 4*/
    int passed;
    textline[4] = '\0';
    for (x=1;x<=(n/4);x++) {
      for (j=x;j<(n-(3*x));j++) {
        for (i=x;i<(n-j-x-x);i++) {
          y = n - x - i - j;
          textline[0] = (char) x;
          textline[1] = (char) j;
          textline[2] = (char) i;
          textline[3] = (char) y;
          passed = 1;
          /* x is the minimum and y=n-x-j-i>x. If j==x, xjiy < jiyx <iyxj */
          /* and xjiy < yxji whether or not i==x. If j, i, and y all >x   */
          /* obviously it is unique minimum. If j>x and i==x the only     */
          /* case to test is xjiy vs iyxj.                                */
          if (i==x) {
            if (y<j) passed = 0;
          }
          if (passed) {
            strcpy(&jumps[ndeads][0],textline);
            ndeads++;
          }
        }
      }
      if ((4*x)==n) {
        memset(textline,x,4);
        strcpy(&jumps[ndeads][0],textline);
        ndeads++;
```

B-8

```
      }
    }
    /*}}}  */
  } else if (m==5) {
    /*{{{  m = 5*/
    int passed;
    textline[5] = '\0';
    for (x=1;x<=(n/5);x++) {
      for (j=x;j<(n-(4*x));j++) {
        for (i=x;i<(n-j-(3*x));i++) {
          for (y=x;y<(n-j-i-x-x);y++) {
            z = n - x - i - j - y;
            textline[0] = (char) x;
            textline[1] = (char) j;
            textline[2] = (char) i;
            textline[3] = (char) y;
            textline[4] = (char) z;
            passed = 1;
            /* x is the minimum and z=n-x-j-i-y>x. The only overlapping     */
            /* cases are xjiyz=xxixz<xzxxi and xjiyz=xjxyz~xyzxj where the  */
            /* places held by x are equal to x and the others >x.           */
            /* Suffices to discard "i==x&&y==x&&j>x" and "i>x&&j>x&&y==x"   */
            /* that is to say "y==x&&j>x".                                  */
            if (y==x) {
              if (x<j) passed = 0;
            }
            if (passed) {
              strcpy(&jumps[ndeads][0],textline);
              ndeads++;
            }
          }
        }
      }
      if ((5*x)==n) {
        memset(textline,x,5);
        strcpy(&jumps[ndeads][0],textline);
        ndeads++;
      }
    }
    /*}}}  */
  } else {
    printf ("m too big or too small.\n");
    exit(1);
  }
  if (mflip) {
    int i, j, bits, errknt;
    errknt = 0;
    for (i=0;i<ndeads;i++) {
      bits = ~steptobit((char *)&jumps[i][0]);
      if (bittostep(bits,(char *)&jumps[i][0])!=mtemp) errknt++;
    }
    printf("%d errors flipping deads and alives.\n",errknt);
    mflip = m;
    m = mtemp;
    mtemp = mflip;
  }
  printf ("%d distinct cases found:",ndeads);
```

B-9

```
x = 0;
for (j=0;j<ndeads;j+=4) {
  printf("\n");
  for(i=j;(i<(j+4))&&(i<ndeads);i++) {
    printf("       %2d",(int)jumps[i][0]);
    for (y=1;y<m;y++) printf(".%2d",(int)jumps[i][y]);
  }
}
printf("\n");
/*}}}  */
/*{{{  test them all (jumps up to ndeads)!*/
if (alive) alive = ((m*D)<65);
if (alive) {
  int success;
  int nups = n - m;
  int upinds[64], downinds[64];
  int downs[18];
  ndim = m*D;
  success = 1;
  for (y=0;y<ndeads;y++) {
    int determinant, invsuccess;
    /*{{{  downs[j] = jth disk down (0 and m at 0)*/
    downs[0] = 0;
    for (i=0;i<m;i++) {
      downs[i+1] = downs[i] + ((int) jumps[y][i]);
    }
    /*}}}  */
    /*{{{  downinds[i] = i-th index on D side (i of matrix)*/
    i = 0;
    for (x=0;x<m;x++) {
      int ind = D*downs[x];
      for (j=0;j<D;j++) {
        downinds[i] = ind + j;
        i++;
      }
    }
    /*}}}  */
    /*{{{  upinds[i] = i-th index on P side (j of matrix)*/
    i = 0;
    j = 0;
    for (x=0;x<n;x++) {
      if (x==downs[j]) {
        j++;
      } else {
        int ind = P*x;
        int k;
        for (k=0;k<P;k++) {
          upinds[i] = ind + k;
          i++;
        }
      }
    }
    if (i!=ndim) {printf("weird counting error\n");exit(1);}
    /*}}}  */
    /*{{{  generate ndim by ndim testmatrix*/
    /* if (ndim>32) { */
      memset((void *)ltestmatrix, 0, 512);
```

B-10

```
        for (j=0;j<ndim;j++) {
          unsigned long long test = ltests[j];
          char *local = &joefile[upinds[j]][0];
          for (i=0;i<ndim;i++) {
            if (local[downinds[i]]=='1') ltestmatrix[i] |= test;
          }
        }
        memcpy((void *)ltestm2, (void *)ltestmatrix, 512);
        determinant = det64();
        if (determinant<1) {
          int rankshort;
          rankshort = 0;
          for (i=0;i<ndim;i++) {
            if (ltestmatrix[i]==(long long)0) rankshort++;
          }
          printf("Rank shortage >= %d\n",rankshort);
        }
        memcpy((void *)ltestmatrix, (void *)ltestm2, 512);
      /*}}}  */
      invsuccess = 0;
      if (determinant) {
          invsuccess = inv64();
      }
      printf(" %2d",downs[0]);
      for (i=1;i<m;i++) printf(".%2d",downs[i]);
      printf(": returned det = %d, inverse-check = %d\n",
        determinant,invsuccess);
      if (determinant!=1) success = 0;
      if (invsuccess!=1) success = 0;
    }
    if (success) {printf("%d,%d test SUCCESSFUL.\n",n,m);
    } else {printf("%d,%d test FAILED.\n",n,m);}
  }
  /*}}}  */
  exit(0);
}
```

B-11

```
#include <stdio.h>
#include <stdlib.h>

#define NMAX 20
#define NUMMATMAX 200

#define NUMDDMAX NMAX
#define NUMCDMAX NMAX
#define MATROWSINMAX NMAX * NUMCDMAX
#define MATSIZEMAX NMAX * NUMDDMAX

main(argc, argv)
int argc;
char *argv[];

{
  extern int gausstest(), gencase(), gn2fast(), gnslow(), init(), showansw();
  extern int gauss();
  int heartcount, heartbeat, matnow, n, m, numdd, numcd, numt, nummat, i;
  int channel_pattern[NUMMATMAX*NMAX], code_channel, code_division, prtinv;
  int data_division, c[(NMAX-2)*NUMDDMAX], ctot[NUMDDMAX], gntype, gninit;
  int numnow, log2num[MATROWSINMAX], totsol, numsol, code, d, tdiv, flip;
  int savetime[MATROWSINMAX], matsize, complete, ctottot, ccdtot[NUMCDMAX];
  long int num[MATROWSINMAX];
  unsigned long int mat[MATSIZEMAX], matinv[MATSIZEMAX], totit, list[NUMMATMAX];

  if (argc < 6) {
    printf("\nCommand line arguments:\nn, m, heartbeat, # of solutions, ");
    printf("getnext type, <t divisor>, <flip>, <print-inv>\n\n");
    return(0);
  }

  n = atoi(argv[1]);
  m = atoi(argv[2]);
  heartbeat = atoi(argv[3]);
  numsol = atoi(argv[4]);
  gninit = atoi(argv[5]);
  gntype = ((gninit == 2) ? 1 : gninit);
  tdiv = ((argc < 7) ? 1 : atoi(argv[6]));
  if (tdiv < 1) tdiv = 1;
  flip = ((argc < 8) ? 0 : atoi(argv[7]));
  prtinv = ((argc < 9) ? 0 : atoi(argv[8]));

  gencase(&code, &d, &numdd, &numcd, &numt, &nummat, channel_pattern, list,
          n, m, tdiv);

  init(&heartcount, &heartbeat, num, &code_channel, &code_division,
       &data_division, c, ctot, &numnow, &totit, &totsol, savetime,
       n, m, numdd, numcd, &matsize, gntype, &ctottot, ccdtot);

  for (;;) {
    if ((totit == 1) && (gninit == 2)) gntype = 0;

    switch (gntype) {

      case 0:
        complete = gnslow(num, &numnow, n, numdd, numcd) * (totit != 0);
```

C-1

```
      break;

    case 1:  case 2:
      complete = gn2fast(num, &code_channel, &code_division, &data_division,
                 c, ctot, &numnow, log2num, savetime, n, m, numdd, numcd);
      if ((totit == 0) && (gninit == 2)) gntype = 0;
      if (flip != 0) {
        num[(n - 1) * numcd - 1] = 1;
        num[(n - 1) * numcd - 2] = 0;
      }

      break;

    case 3:
      complete = gn3fast(num, &code_channel, &code_division, &data_division,
          c, ctot, &numnow, log2num, savetime, n, m, numdd, numcd, &ctottot,
          ccdtot);
      break;

    case 4:
      complete = gn3fast(num, &code_channel, &code_division, &data_division,
          c, ctot, &numnow, log2num, savetime, n, m, numdd, numcd, &ctottot,
          ccdtot);
      for (i = 0; (num[0] & (1 << i)) && i < (numdd-1); i++);
      num[0] |= 1 << i;
      break;

  }

  if (complete) break;

  matnow = 0;
  while (matnow != nummat) {
    makemat(num, channel_pattern, mat, matnow, n, m, numdd, numcd);
    if (gausstest(mat, matsize) == 0) matnow++;
    else break;
  }

  totit++;
  if (matnow == nummat) {
    totsol++;
    showansw(1, num, totsol, totit, matnow, n, m, numdd, numcd, nummat, tdiv);
    if (prtinv != 0) for (i = 0; i < nummat; i++) {
      makemat(num, channel_pattern, mat, i, n, m, numdd, numcd);
      gauss(mat, matinv, matsize, i);
    }

    if (totsol == numsol) break;
  }

  if (heartcount != heartbeat) heartcount++;
  else {
    showansw(0, num, totsol, totit, matnow, n, m, numdd, numcd, nummat, tdiv);
    heartcount = 1;
  }
}
```

C-2

```
  /*  Execution is complete */
  if (totsol != numsol) showansw(-1, num, totsol , totit, matnow, n, m, numdd,
                                  numcd, nummat, tdiv);
  return(0);
}

int gencase(c, d, numdd, numcd, numt, nummat, channel_pattern, list, n, m, tdiv)
int n, m, *c, *d, *numdd, *numcd, *numt, *nummat, tdiv;
int channel_pattern[NUMMATMAX*NMAX];
unsigned long int list[NUMMATMAX];

{
  unsigned long int imin, inew, ishift, irot, k, nbound, nmask;
  int i, j, lcm, missing, received;

  for (i = 1, lcm = 1; i <= n; i++)
   if ((lcm-(lcm/i)*i) != 0) {
     for (j = 2; (j*lcm-(((j*lcm)/i)*i)) != 0; j++);
     lcm *= j;
   }

  *d = lcm/n;
  *c = *d*m/(n-m);
  *numdd = n - m;
  *numcd = m;
  i = ((*numdd < *numcd) ? *numdd : *numcd);
  for (j = i; j > 1; j--) {
    if (   ( (*numdd - ((*numdd/j)*j) ) == 0 ) &&
           ( (*numcd - ((*numcd/j)*j) ) == 0 )   ) {
      *numdd /= j;
      *numcd /= j;
    }
  }
  *numt = *d/(*numdd)*tdiv;
  *numdd *= tdiv;
  *numcd *= tdiv;

  printf("\n\nn = %i, m = %i, d = %i, c = %i, ", n, m, *numdd, *numcd);
  printf("p = %i\n", tdiv);

  nbound = 1 << n;
  nmask = nbound - 1;
  *nummat = 0;

  for (i = 0; i < nbound; i++) {
    for (k = 0, inew = i; inew != 0; inew >>= 1) if (inew & 01) ++k;
    if (k == m) {
      imin = inew = i;
      for (j = 1; j < n; j++) {
        ishift = inew << 1;
        inew = (ishift & nmask) + ((ishift & nbound) != 0);
        if (imin > inew) imin = inew;
      }
      for (j = 0; j < *nummat; j++) if (imin == list[j]) break;
      if (j == *nummat) list[(*nummat)++] = imin;
    }
  }
```

C-3

```
  for (i = 0; i < *nummat; i++) {
    imin = list[i];
    received = 0;
    missing = n - m;
    for (j = 0, irot = 1 << (n-1); j < n; j++, irot >>= 1)
      channel_pattern[(i * n) + ((imin & irot) ? missing++ : received++)] = j;
  }

  printf("\nThe following %i unique cases are to be tested:", *nummat);
  for (i = 0; i < *nummat; i++) {
    printf("\n%u> %lu::    ", (i+1), list[i]);
    for (j = 0; j < n; j++) {
      if (j == n - m) printf ("  ");
      printf("%i ", channel_pattern[(i * n) + j]);
    }
  }

  printf("\n\nThe first pattern generated is:");

}

int init(heartcount, heartbeat, num, code_channel, code_division,
 data_division, c, ctot, numnow, totit, totsol, savetime,
 n, m, numdd, numcd, matsize, gntype, ctottot, ccdtot)
int *heartcount, *heartbeat, n, m, numdd, numcd, *matsize;
long int num[];
int *code_channel, *code_division, *data_division, c[];
int ctot[], *numnow, *totsol, savetime[], gntype, *ctottot, ccdtot[];
unsigned long int *totit;

{
  int i, j;

  *totit = 0;
  *totsol = 0;

  *ctottot = 0;

  *heartcount = *heartbeat;

  switch (gntype) {

    case 3: case 4:
      *numnow = ((n - 1) * numcd) - 1;
      *data_division = -2;
      *code_channel = n - 2;
      break;

    default:
      *numnow = ((n - 2) * numcd) - 1;
      *data_division = -1;
      *code_channel = n - 3;
      break;
  }

  *code_division = numcd - 1;
  *matsize = m * numdd;
```

C-4

```
  for (i = 0; i < numdd; i++) {
    ctot[i] = -m;
    for (j = 0; j < n - 2; j++)  c[(j * numdd) + i] = 0;
  }
  for (i = 0; i < numcd; i++) ccdtot[i] = 0;
  for (i = 0; i < ((n - 1) * numcd); i++) num[i] = (savetime[i] = 0);

}

int gnslow(num, numnow, n, numdd, numcd)
long int num[MATROWSINMAX];
int *numnow, n, numdd, numcd;

{
  *numnow = 0;
  for(;;) {    /* Loop forever                                            */
    if (num[*numnow] != (1<<numdd)-1) {          /* If num is not too high */
      ++num[*numnow];                              /* Update num           */
      if (*numnow==0) return(0);    /* If this is the lowest numnow, return  */
      else num[--*numnow] = -1; /* Else this is not lowest numnow, so update */
    }                              /* End If num is not too high           */
    else {                                       /* Else num is too high   */
      if (*numnow == ((n - 1) * numcd) - 1) return(1);     /* Final return  */
      else ++*numnow;                             /* Go to a higher numnow */
    }                          /* End Else data division is too high       */
  }                          /* End of Loop forever                        */
}                          /* End of main                                  */

int gn2fast(num, code_channel, code_division, data_division, c, ctot, numnow,
            log2num, savetime, n, m, numdd, numcd)
long int num[MATROWSINMAX];
int *code_channel, *code_division, *data_division, *numnow;
int savetime[MATROWSINMAX], c[(NMAX-2)*NUMDDMAX], ctot[NUMDDMAX];
int log2num[MATROWSINMAX], n, m, numdd, numcd;

{
  if (*data_division >= 0) {
    --c[(*code_channel * numdd) + *data_division];        /* Decrement c     */
    --ctot[*data_division];                         /* Decrement ctot        */
  }

  for(;;) {    /* Loop forever                                            */
    if (++*data_division < numdd) {  /* If new data division is not too high */
      /* If current position is OK, mark it as used and possibly do updates  */
      if ( (savetime[*numnow]==0) || (ctot[*data_division] != -m) ) {
        if (ctot[*data_division] == -m) savetime[*numnow] = 1;
        /* If current choice meets tests, do updates                        */
        if ( (ctot[*data_division] < 0)
         && (c[(*code_channel * numdd) + *data_division]==0) ) {
          num[*numnow] = 1 << *data_division;           /* Update num        */
          log2num[*numnow] = *data_division + 1;       /* Update log2num    */
          ++c[(*code_channel * numdd) + *data_division];      /* Increment c */
          ++ctot[*data_division];                       /* Increment ctot   */
          if (*numnow==0) return(0);/* If this is the lowest numnow, return  */
          else {                   /* Else this is not the lowest numnow    */
            --*numnow;                               /* Decrement numnow    */
```

C-5

```
          *data_division = -1;            /* Adjust to new data division   */
          savetime[*numnow] = 0; /*Reset to virgin position not yet found  */
          if (--*code_division < 0) { /* If new code division is too low   */
            *code_division = numcd - 1;  /* Reset code division            */
            --*code_channel;            /* Decrement code channel          */
          }                           /* End If code division is too low   */
        }                           /* End Else this is not lowest numnow  */
      }                           /* End If current choice meets tests     */
    }                           /* End If current position is OK           */
  }                           /* End If data division is not too high      */
  else {                               /* Else data division is too high   */
    if (*numnow == (n - 3) * numcd)           /* If everything checked */
      return(1);                                         /* Final return */
    else {                                    /* Go to a higher numnow */
      *data_division=log2num[++*numnow]-1;    /* Update data_div. & numnow */
      if (++*code_division==numcd) {   /* If new code division is too high */
        *code_division = 0;                  /* Reset code division        */
        ++*code_channel;                     /* Increment code channel     */
      }                            /* End If new code division is too high */
      --c[(*code_channel * numdd) + *data_division];      /* Decrement c    */
      --ctot[*data_division];                /* Decrement ctot             */
    }                            /* End If this is not the highest numnow */
  }                                 /* End Else data division is too high */
 }                                  /* End of Loop forever                 */
}                                   /* End of main                         */

int gn3fast(num, code_channel, code_division, data_division, c, ctot, numnow,
            log2num, savetime, n, m, numdd, numcd, ctottot, ccdtot)
long int num[MATROWSINMAX];
int *code_channel, *code_division, *data_division, *numnow;
int savetime[MATROWSINMAX], c[(NMAX-2)*NUMDDMAX], ctot[NUMDDMAX];
int log2num[MATROWSINMAX], n, m, numdd, numcd, *ctottot, ccdtot[NUMCDMAX];

{
  int i, nmmax;

  nmmax = (m > (n - m) ? (m) : (n - m));

  if (*data_division >= 0) {
    --c[(*code_channel * numdd) + *data_division];        /* Decrement c      */
    --ctot[*data_division];                          /* Decrement ctot       */
    --ccdtot[*code_division];
    --*ctottot;
  }

  for(;;) {                                               /* Loop forever */
    if (++*data_division == numdd||savetime[*numnow]==1) { /* If need higher */
      if (*numnow == (n - 1) * numcd - 1)           /* If everything checked */
        return(1);                                         /* Final return */
      else {                                      /* Go to a higher numnow */
        *data_division=log2num[++*numnow];     /* Update data_div. & numnow */
        if (++*code_division==numcd) {   /* If new code division is too high */
          *code_division = 0;                      /* Reset code division */
          ++*code_channel;                      /* Increment code channel */
        }                            /* End If new code division is too high */
        if (*data_division >= 0) {
          --c[(*code_channel * numdd) + *data_division];      /* Decrement c */
```

C-6

```
      --ctot[*data_division];                          /* Decrement ctot */
      --ccdtot[*code_division];
      --*ctottot;
    }
  }                           /* End If this is not the highest numnow */
}                                /* End If data division is too high */
else {                        /* If new data division is not too high */
  /* If current position is OK, mark it as used and possibly do updates  */
  if (*data_division<0||savetime[*numnow]==0||ctot[*data_division]!=-m) {
    if (*data_division>=0 &&ctot[*data_division]==-m) savetime[*numnow] = 1;
                          /* If current choice meets tests, do updates */
    if (*data_division < 0 || ((ctot[*data_division] < 0)
     && (c[(*code_channel * numdd) + *data_division]==0))) {
      num[*numnow] = (1 << (*data_division + 1)) >> 1;     /* Update num */
      log2num[*numnow] = *data_division;               /* Update log2num */
      if (*data_division >= 0) {
        ++c[(*code_channel * numdd) + *data_division];    /* Increment c */
        ++ctot[*data_division];                        /* Increment ctot */
        ++ccdtot[*code_division];
        ++*ctottot;
      }
      if (*numnow!=0) {
        *data_division = -2;             /* Adjust to new data division */
        savetime[--*numnow]=0; /* Indicate unused position not yet found */
        if (--*code_division < 0) {   /* If new code division is too low */
          *code_division = numcd - 1;           /* Reset code division */
          --*code_channel;                   /* Decrement code channel */
          if (*ctottot < (numdd * m * (nmmax - 1 - *code_channel)/nmmax))
           savetime[*numnow] = 1;
          for (i = numcd - 1; i >= 0; i--)
           if ((ccdtot[i] > m*numdd/numcd) ||
            (ccdtot[i] < m*numdd/numcd - 1 - *code_channel))
             savetime[*numnow] = 1;
        }                            /* End If code division is too low */
      }                            /* End If this is not lowest numnow */
      else if (*ctottot == numdd*m) {
        for (i = numcd - 1; i >= 0; i--) if (ccdtot[i] > m*numdd/numcd) {
          savetime[*numnow] = 1;
          break;
        }
        if (i < 0) return(0);        /* return for now */
        else if (*data_division >= 0) {
          --c[(*code_channel * numdd) + *data_division];  /* Decrement c */
          --ctot[*data_division];                      /* Decrement ctot */
          --ccdtot[*code_division];
          --*ctottot;
        }
      }
      else if (*data_division >= 0) {
        --c[(*code_channel * numdd) + *data_division];    /* Decrement c */
        --ctot[*data_division];                        /* Decrement ctot */
        --ccdtot[*code_division];
        --*ctottot;
      }
    }                           /* End If current choice meets tests */
  }                                /* End If current position is OK */
}                           /* End Else data division is not too high */
```

C-7

```
  }                                                /* End of Loop forever */
}                                                       /* End of main */

makemat (num, channel_pattern, mat, matnow, n, m, numdd, numcd)
long int num[MATROWSINMAX];
unsigned long int mat[MATSIZEMAX];
int channel_pattern[NUMMATMAX*NMAX], matnow, n, m, numdd, numcd;

{
  int i, j, k, p, x;
  unsigned long int y;

  for (i = 0; i < (n - m); i++) {
    x = channel_pattern[(matnow * n) + i];
    for (j = 0; j < numcd; j++) {
      y = 0;
      for (p = (matnow * n) + (n - m); p < (matnow * n) + n; p++) {
        k = j + (n - 1) * numcd + (numcd * (x - channel_pattern[p]));
        while (k < 0) k += (n * numcd);
        while (k >= (n * numcd)) k -= (n * numcd);
        y = (y << numdd) | num[k];
      }
      mat[(i*numcd)+j] = y;
    }
  }
  return;
}

int gausstest(mat, matsize)
unsigned long int mat[MATSIZEMAX];
int matsize;
{
  int i, j, k;
  unsigned long int r;

  for (i=k=0, r = 1 << (matsize-1); i < matsize; k = ++i, r>>=1){ /* Columns */
    if (mat[i] < r) {   /* If diagonal point != 1, find row with 1 in column */
      for (k = i + 1; (k < matsize) && (mat[k] < r); k++);        /* Looking */
      if (k == matsize) return(1);        /* Return if all zeroes in a column */
      mat[k] ^= mat[i] ^= mat[k];                      /* Modulo-2 row switch! */
    }        /* after this statement, diagonal point is guaranteed to be a 1 */
    for (j=0; j<i; j++) if ((mat[j]&r) != 0) mat[j] ^= mat[i];   /* Clean col */
    for (j = k+1; j < matsize; j++) if (mat[j] >= r) mat[j] ^= mat[i];  /* " */
  }                                           /* End of going through columns */
  return(0);
}      /* End of gaussian elimination test for a modulo-2-invertable matrix */

gauss(mat, matinv, matsize, matnow)
unsigned long int mat[MATSIZEMAX], matinv[MATSIZEMAX];
int matsize, matnow;
{
  int i, j, k;
  unsigned long int r;

  printf("\nCase %i original matrix\n", (matnow + 1));
  for (i = 0; i < matsize; i++) {
```

C-8

```
      for (j = matsize - 1; j >= 0; j--) printf("%i ", (mat[i] & (1 << j)) ? 1 :
0);
      printf("\n");
    }

    for (i = 0, r = 1 << (matsize-1); i < matsize; i++, r >>= 1 ) matinv[i] = r;

    for (i=k=0, r = 1 << (matsize-1); i < matsize; k = ++i, r>>=1){ /* Columns */
      if (mat[i] < r) {   /* If diagonal point != 1, find row with 1 in column */
        for (k = i + 1; (k < matsize) && (mat[k] < r); k++);          /* Looking */
        if (k == matsize) return;          /* Return if all zeroes in a column */
        mat[k] ^= mat[i] ^= mat[k];                       /* Modulo-2 row switch! */
        matinv[k] ^= matinv[i] ^= matinv[k];   /* Switch rows in shadow matrix */
      }       /* After this statement, diagonal point is guaranteed to be a 1 */
      for (j=0; j<i; j++) if ((mat[j] & r) != 0) {  /* Rows above the diagonal */
        mat[j] ^= mat[i];      /* Clean column to get rid of 1, except diagonal */
        matinv[j] ^= matinv[i];    /* Corresponding operation in shadow matrix */
      }
      for (j = k+1; j < matsize; j++) if (mat[j] >= r) { /* Rows below diagonal*/
        mat[j] ^= mat[i];      /* Clean column to get rid of 1, except diagonal */
        matinv[j] ^= matinv[i];    /* Corresponding operation in shadow matrix */
      }
    }                                           /* End of going through columns */

    printf("\nCase %i inverted matrix\n", (matnow + 1));
    for (i = 0; i < matsize; i++) {
      for (j = matsize-1; j >= 0; j--) printf("%i ", (matinv[i]&(1<<j)) ? 1 : 0);
      printf("\n");
    }

    return;
}   /* End of gaussian elimination method to invert a 0-1 matrix in modulo-2 */

int showansw(showtype, num, totsol, totit, matnow, n, m, numdd, numcd, nummat,
             tdiv)
long int num[MATROWSINMAX];
int showtype, totsol, matnow, n, m, numdd, numcd, nummat, tdiv;
unsigned long int totit;

{
    int i, j, k, p, x;

    if (showtype != 0) {
      printf("\nn = %i, m = %i, d = %i, c = %i, ", n, m, numdd, numcd);
      printf("p = %i, cases = %i", tdiv, nummat);
    }

    if (showtype > 0) printf ("\n\nSolution %i has been found:", totsol);

    if (showtype < 0) printf("\n\nThe last pattern checked was:");

    printf ("\n%lu  %i:  ", totit, matnow);

    for (i = 0; i < (n - 1); i++) {
      for (j = 0; j < numcd; j++) printf ("%lu ", num[(i*numcd)+j]);
      printf (" ");
    }
```

C-9

```
  printf("\n");

  if (showtype > 0) {
    printf ("\n");
    for (i = p = 0; i < (n - 1); i++)
     for (j = 0; j < numcd; j++) {
       ((j==0) ? printf("            ") : printf("            "));
       for (k = (numdd - 1); k >= 0; k--) {
         p += x = ((num[(i * numcd) + j] & (1 << k)) ? 1 : 0);
         printf("%i", x);
       }
       ((j==0) ? printf(" -\n") : printf("\n"));
     }
    printf("\nSolution %i contains %i ones; ", totsol, p);
    printf("the lower theoretical bound is %i ones\n\n", m*numdd);
  }

  if (showtype < 0) switch (totsol) {
    case 0:  printf("\nNo solutions were found\n"); break;
    case 1:  printf("\nA total of 1 solution was found\n"); break;
    default: printf("\nA total of %i solutions were found\n", totsol);
  }

}
```

C-10

APPENDIX D

DUALITY THEOREM for Wiencko codes

The following is an additional technique for finding Wiencko arrays and codes for those various applications. The "duality" to be described gives a one to one relationship between(n, m) Wiencko arrays and (n, n-m) Wiencko arrays. This means if you have an (n, m) Wiencko code, you can automatically get an (n, n-m) Wiencko code, and vice versa.

DUALITY

Let W be a j by k array of H by Q matrices. We define the "transpose" Wt of W as the k by j array of Q by H matrices such that, for every z, w such that 0<=z<j and 0<=w<k,

$$(1)\quad Wt_{wz} = (W_{zw})^{t}$$

It follows trivially from this that the equivalent matrix to Wt is the transpose of the equivalent matrix to W. In discussions below, we will let z'w' be the array subscripts of Wt, so that z'=w and w'=z in (1).

THEOREM (Duality):

W is an (n, m) Wiencko array if and only if Wt is an (n, n-m) Wiencko array.

Proof:

It suffices to prove it in one direction (assuming W is Wiencko and proving Wt is), since exchanging m for n-m in the proof then proves it in the other direction.

D-1

It is trivial to show that if H and Q satisfy the test n/m = (H/Q)+1, then H'=Q and Q'=H satisfy the test for (n, m') where m'=n-m.

All that remains, therefore, is to prove that the "invertibility" conditions are satisfied. But there is a one to one correspondence between the subsets S of m disks out of n, and the subsets S' of m'=n-m disks out of n: let S' be all the disks not in S. The subarray of W corresponding to S consists of all matrix entries with z in S' and w in S. But the subarray of Wt corresponding to S' consists of all matrix entries with z' in S and w' in S'. By (1) these correspond exactly, but transposed; so that the equivalent matrix to the subarray of Wt corresponding to S' is the transpose of the equivalent matrix to the subarray of W corresponding to S.

If a matrix is invertible, so is its transpose. Since this holds for all possible values of S', it is proven that Wt satisfies the invertibility conditions. This completes the proof.

Corollary:
The one to one duality relation maps zero diagonal (n, m) Wiencko arrays into zero diagonal (n, n-m) Wiencko arrays, and maps rotationally symmetric (n, m) Wiencko arrays into rotationally symmetric (n, n-m) Wiencko arrays.

Proof:
Trivial consequence of the definitions. The final step is to describe how to exchange the standard descriptions. If the standard description of an (n,m) Wiencko code is

M1

D-2

```
M2

.
.
.

M(n-1)
```

where the M's are H by Q bit matrices, then it trivially follows that the standard description of its dual is

$$M(n-1)^t$$

```
.
.
.
```

$$M2^t$$
$$M1^t$$

EXAMPLE

The (9,4) solution has the following standard description:

```
9 4 5 4
01000 -
00001
10000
00010
00000 -
00000
00001
10000
00010 -
01000
00100
10000
00010 -
10000
00100
01000
01000 -
00100
00000
00001
00100 -
```

D-3

```
00010
00001
00000
00000 -
00000
00000
00000
00000 -
00000
00000
00000
```

By applying the duality, the following (9,5) code can be found:

```
9 5 4 5
0000 -
0000
0000
0000
0000
0000 -
0000
0000
0000
0000
0000 -
0000
1000
0100
0010
0000 -
1000
0100
0000
0001
0100 -
0001
0010
1000
0000
0001 -
0100
0010
1000
0000
0001 -
0000
0000
0000
0010
0010 -
1000
```

D-4

```
0000
0001
0100
```

This code was confirmed as a Wiencko code. Other lacunae that can be filled by this method are (10,7), (13,11), (15,13), (16,14), (17,15),(18,16) and (18,15).

D-5

```
/*{{{  includes*/
/* #include <sys/types.h>
   #include <sys/stat.h>
   #include <fcntl.h>
   #include <termios.h> */
#include <stdio.h>
#include <stdlib.h>
/*}}}  */
/*{{{  defines*/

#define DEBUG 1
#define MAX_JDIM 100
#define MAX_IDIM 306
#define MAX_LENGTH 8192
#define MAX_COUNT  1024
#define FILENAME "file.joe"
#define FILEOUT "file.ljd"
/*}}}  */
main(int argc, char *argv[])
{
  FILE *stream;
  char textline[512],filename[128],rotated[10],fileout[128];
  char joefile[MAX_JDIM] [MAX_IDIM];
  int n,m,D,P,datalines,i,j,x,y,z,ndeads,alive,ndim,mtemp,mflip;
  /* START OF THE STATEMENTS */
  if (argc>1) {
    strcpy(filename,argv[1]);
  } else {
    strcpy(filename,FILENAME);
  }
  if (argc>2) {
    strcpy(fileout,argv[2]);
  } else {
    strcpy(fileout,FILEOUT);
  }
  printf ("\n\nTEST PROGRAM dualize READS %s...\n\n\n", filename);
  alive = 1;
  /*{{{  build the D to P matrix*/

  stream = fopen( filename, "r");
  if (stream==NULL) {
    printf ("%s does not exist.\n",filename);
    exit(1);
  }
  printf ("%s exists. Reading in its data.\n",filename);
  if (fgets(textline,511,stream)!=NULL) {
    sscanf(textline,"%d %d %d %d",&n,&m,&D,&P);
    printf("n=%d,m=%d,D=%d,P=%d\n",n,m,D,P);
    if (n&&m&&D&&P&&(n>m)&&((n*P)==((D+P)*m))) {
      printf("Confirmed correct ratio.\n");
    } else {printf("Error in ratio or zero entry.\n");exit(1);}
  }
  mtemp = n-m;
  if ((n>18)||((m>5)&&(mtemp>5))||((n*P)>MAX_JDIM)||((n*D)>MAX_IDIM)) {
    printf("Horsefeathers! Too big for our buffers! ixj=%dx%d\n",n*D,n*P);
    if ((n>18)||((m>5)&&(mtemp>5))) exit(1);
    alive = 0;
```

D-6

```
  }
  if (alive) {
    /*{{{  read in and check the data*/
    memset(joefile, (int)'0', MAX_JDIM*MAX_IDIM);
    datalines = 0;
    memset(textline,(int)'7',D);
    while (fgets(textline,511,stream)!=NULL) {
      printf(">>>%s",textline);
      for (x=0;x<D;x++) {
        if ((((int)textline[x])-48)>>1) {
          printf("Illegal character in data line %d.\n",datalines);
          alive = 0;
        }
      }
      memcpy(&joefile[datalines+P][0],textline,D);
      memset(textline,(int)'7',D);
      datalines++;
    }
    printf("Total of %d lines of data.",datalines);
    if (datalines==((n-1)*P)) {printf(" Correct.\n");
    } else {printf(" Should have been %d.\n",(n-1)*P);alive = 0;}
    /*}}}  */
  }
  fclose(stream);
  if (alive) {
    /*{{{  write the transposed data*/
    {
      int md = n - m;
      int Pd = D;
      int Dd = P;
      int ndisk, id, jd;
      stream = fopen( fileout, "w");
      if (stream==NULL) {
        printf ("%s failed to open.\n",fileout);
        exit(1);
      }
      printf ("%s is open for writing.\n",fileout);
      fprintf(stream,"%d %d %d %d\n",n,md,Dd,Pd);
      for (ndisk=1;ndisk<n;ndisk++) {
        int nofs = (n-ndisk)*Dd;
        int i,j;
        for (j=0;j<Pd;j++) {
          for (i=0;i<Dd;i++) textline[i] = joefile[nofs+i][j];
          textline[Dd] = '\0';
          if (j) {fprintf(stream,"%s\n",textline);
          } else {fprintf(stream,"%s -\n",textline);}
        }
      }
      fclose(stream);
      printf ("%s is complete.\n",fileout);
    }
    /*}}}  */
  }
  /*}}}  */
  exit(0);
}
```

D-7

CLAIMS

1. A data storage apparatus, comprising:

a plurality of n data storage devices;

data comprising a plurality of n data groupings;

each one of said n data groupings stored in one of said n data storage devices;

each one of said n data groupings comprising a data portion and a data redundancy portion; and

said n data portions being recoverable from any and all combinations of n-m data grouping(s) on n-m data storage device(s) when the other m data grouping(s) are unavailable, where $1 < m < n$.

2. A data storage apparatus according to claim 1, further comprising:

a storage controller; and

said storage controller operative to recover said n data portions from any and all combinations of n-m data grouping(s) on n-m disk(s) when the other m data grouping(s) are unavailable.

3. A method for recovering data, comprising:

receiving a plurality of n-m data grouping(s) of a plurality of n data groupings of data, where $1 < m < n$, each n data grouping comprising a data portion and a redundancy portion; and

recovering the n data groupings from the plurality of n-m data groupings when the other m data groupings are not available.

4. A disk array having n disks, comprising:

a stripe set stored across the n disks;

each stripe of the stripe set comprising a data portion of H data chunks and a redundancy portion of Q redundancy chunks;

a relationship existing between the data portions and the redundancy portions;

the relationship being based on an n*H by n*Q bit matrix, the bit matrix being such that:

the bit matrix is representible by an n by n array of H by Q bit submatrices, where n/m = (H/Q) + 1;

the bit matrix has a plurality of n!/(m!*(n-m)!) composite bit submatrices definable therefrom, each such composite bit submatrix being definable from bit submatrices at the intersection of a unique selection of m column(s) of the n by n array and a unique selection of (n-m) row(s) of the n by n array that correspond to those (n-m) column(s) not included in the unique selection of m column(s), where each one of the composite submatrices is invertible; and

the relationship between the data portions and the redundancy portions being such that each one of Q redundancy chunks is the exclusive-OR of the n*H bits of the data portions and the n*H bits in the row of the bit matrix associated with such redundancy bit.

5. An apparatus suitable for use in generating redundancy data, comprising:

a storage medium; and

codes embedded in said storage medium, said codes being representible by an n*H by n*Q bit matrix, the bit matrix being representible by an n by n array of H by Q bit submatrices, where $n/m = (H/Q) + 1$, the bit matrix having a plurality of $n!/(m!*(n-m)!)$ composite submatrices definable therefrom, each composite submatrix being definable from submatrices at the intersection of a unique selection of m column(s) of the n by n array and a unique selection of (n-m) row(s) of the n by n array that correspond to those (n-m) column(s) not included in the unique selection of m column(s), each such composite submatrix being invertible.

6. A programmable gate array (PGA), comprising:

logic gates configured to execute a set of XOR-based functions;

the set of XOR-based functions being representable by an n*H by n*Q bit matrix, the bit matrix being such that:

the bit matrix is representible by an n by n array of H by Q bit submatrices, where n/m = (H/Q) + 1;

the bit matrix has a plurality of n!/(m!*(n-m)!) composite submatrices definable therefrom, each composite submatrix being definable from submatrices at the intersection of a unique selection of m column(s) of the array and a unique selection of (n-m) row(s) of the array that correspond to those (n-m) column(s) not included in the unique selection of m column(s); and

each composite submatrix being invertible.

7. One or more data storage devices, comprising:

a plurality of n storage location areas;

each one of the n storage location areas having one of n data groupings stored thereat;

each one of the n data groupings comprising a data portion of H data chunks and a redundancy portion of Q redundancy chunks,

a relationship existing between the data portions and the redundancy portions;

the relationship being based on an n*H by n*Q bit matrix, the bit matrix being such that:

the bit matrix is representible by an n by n array of H by Q bit submatrices, where n/m = (H/Q) + 1;

the bit matrix has a plurality of n!/(m!*(n-m)!) composite bit submatrices definable therefrom, each such composite bit submatrix being definable from bit submatrices at the intersection of a unique selection of m column(s) of the n by n array and a unique selection of (n-m) row(s) of the n by n array that correspond to those (n-m) column(s) not included in the unique selection of m column(s), where each one of the composite submatrices is invertible; and

the relationship between the data portions and the redundancy portions being such that each one of Q redundancy chunks is the exclusive-OR of the n*H bits of the data portions and the n*H bits in the row of the bit matrix associated with such redundancy bit.

8. A redundancy data generation apparatus operative to execute one or more functions from a set of n redundancy data generation functions, comprising:

the set of n redundancy data generation functions being representible by a bit matrix of dimensions n*H by n*Q;

the bit matrix being representible by an n by n array of H by Q bit submatrices, where n/m = (H/Q) + 1;

the bit matrix having a plurality of n!/(m!*(n-m)!) composite submatrices definable therefrom;

each composite submatrix being definable from the intersection of a unique selection of m column(s) of the n by n array and a unique selection of (n-m) row(s) of the array that correspond to those (n-m) column(s) not included in the unique selection of m column(s); and

each composite submatrix being invertible.

9. A method of generating redundancy data with a set of n redundancy data generation functions, the set of n redundancy data generation functions being representible by an n*H by n*Q bit matrix, the bit matrix being representible by an n by n array of H by Q bit submatrices, where n/m = (H/Q) + 1, the bit matrix having a plurality of n!/(m!*(n-m)!) composite submatrices definable therefrom, each composite submatrix being definable from the intersection of a unique selection of m column(s) of the n by n array and a unique selection of (n-m) row(s) of the array that correspond to those (n-m) column(s) not included in the unique selection of m column(s), each composite submatrix being invertible, the method comprising:

receiving a plurality of n data groupings; and

executing the set of n data redundancy functions on the plurality of n data groupings to thereby generate a plurality of n redundancy data groupings associated therewith.

10. A method for use in providing adjustable disk loss insurance, comprising:

receiving control data indicative of an amount of disk redundancy desired for the disk array;

selecting a set of data redundancy functions based on the control data; and

generating redundancy data based on user data and the selected set of data redundancy functions.

11. A programmable gate array (PGA) operative to provide for adjustable data redundancy, comprising:

logic gates configured to selectively execute one of at least a first set of XOR-based functions and a second set of XOR-based functions;

the first set of XOR-based functions being representable by a first bit submatrix having dimensions of $n*H_1$ by $n*Q_1$, the first bit matrix being such that:

the first bit matrix is representible by a first n by n array of H_1 by Q_1 bit submatrices, where $n/m_1 = (H_1/Q_1) + 1$, and $1 < m_1 < n$;

the first bit matrix has a first plurality of $n!/(m_1!*(n-m_1)!)$ composite submatrices definable therefrom, each composite submatrix of the first plurality being definable from the intersection of a unique selection of m_1 column(s) of the first n by n array and a unique selection of $(n-m_1)$ row(s) of the first n by n array that correspond to those $(n-m_1)$ column(s) not included in the unique selection of m_1 column(s), where each composite submatrix of the first plurality is invertible;

the second set of XOR-based functions being representable by a second bit matrix having dimensions of $n*H_2$ by $n*Q_2$, the second bit matrix being such that:

the second bit matrix is representible by a second n by n array of H_2 by Q_2 bit submatrices, where $n/m_2 = (H_2/Q_2) + 1$, and $m_1 < m_2 < n$; and

the second bit matrix has a second plurality of $n!/(m_2!*(n-m_2)!)$ composite submatrices definable therefrom, each composite submatrix of the second plurality being definable from the intersection of a unique selection of m_2 column(s) of the second n by n array and a unique selection of $(n-m_2)$ row(s) of the second n by n array that correspond to those $(n-m_2)$ column(s) not included in the unique selection of m_2 column(s), where each composite submatrix of the second plurality is invertible.

12. A method for use in a radio frequency (RF) communication system, the method involving data comprising a plurality of n data groupings respectively associated with a plurality of n parity groupings, where n and m are integers and $1<m<n$, the method comprising:

in a first communication device:

modulating radio frequency (RF) signals with the data and providing transmission over a plurality of n channels, each channel being associated with one of the n data and parity groupings;

in a second communication device:

receiving and demodulating the modulated RF signals to regenerate the data, which may have errors from interference;

determining whether and which n data groupings are adversely affected with errors; and

recovering the data from (n-m) data groupings when the other m data groupings are determined to be adversely affected with errors.

13. The method according to claim 12, wherein the plurality of n channels are defined by n different RF frequencies.

14. The method according to claim 12, wherein the plurality of n channels are defined by n different time slots of a single RF frequency.

15. The method according to claim 12, wherein the plurality of n channels are defined by n different code divisions.

16. A method for use in a computer network having at least n servers, the method involving data and parity spread across the n servers, where m and n are integers and 1<m<n, the method comprising:

sending a data request to the n servers;

receiving a response from the first (n-m) of n servers;

selecting a data recovery function based on which (n-m) of n servers first responded; and

recovering the data based on the data and parity received from the (n-m) servers that first responded and the selected data recovery function.

17. The method according to claim 16, where m = n-1

18. A computer-implemented method for use in determining whether data representing a candidate bit matrix is a Wiencko bit matrix, the candidate bit matrix having dimensions of n*H by n*Q and being representible by an n by n array of H by Q submatrices, where n, m, H, and Q are positive integers, 1 < m < n, and n/m = (H/Q) + 1, the method comprising:

selecting an H*m by Q*(n-m) composite submatrix from the candidate bit matrix, the composite submatrix formed by submatrices from the intersection of a unique selection of m column(s) of the array and a unique selection of (n-m) row(s) of the array that correspond to those (n-m) column(s) not included in the unique selection of m column(s);

determining whether the selected composite submatrix is invertible;

repeating the selecting and testing as needed to determine whether all n!/(m!*(n-m)!) composite submatrices of the candidate bit matrix are invertible; and

providing an indication as to whether or not all n!/(m!*(n-m)!) composite submatrices are invertible based on the selecting, testing, and repeating.

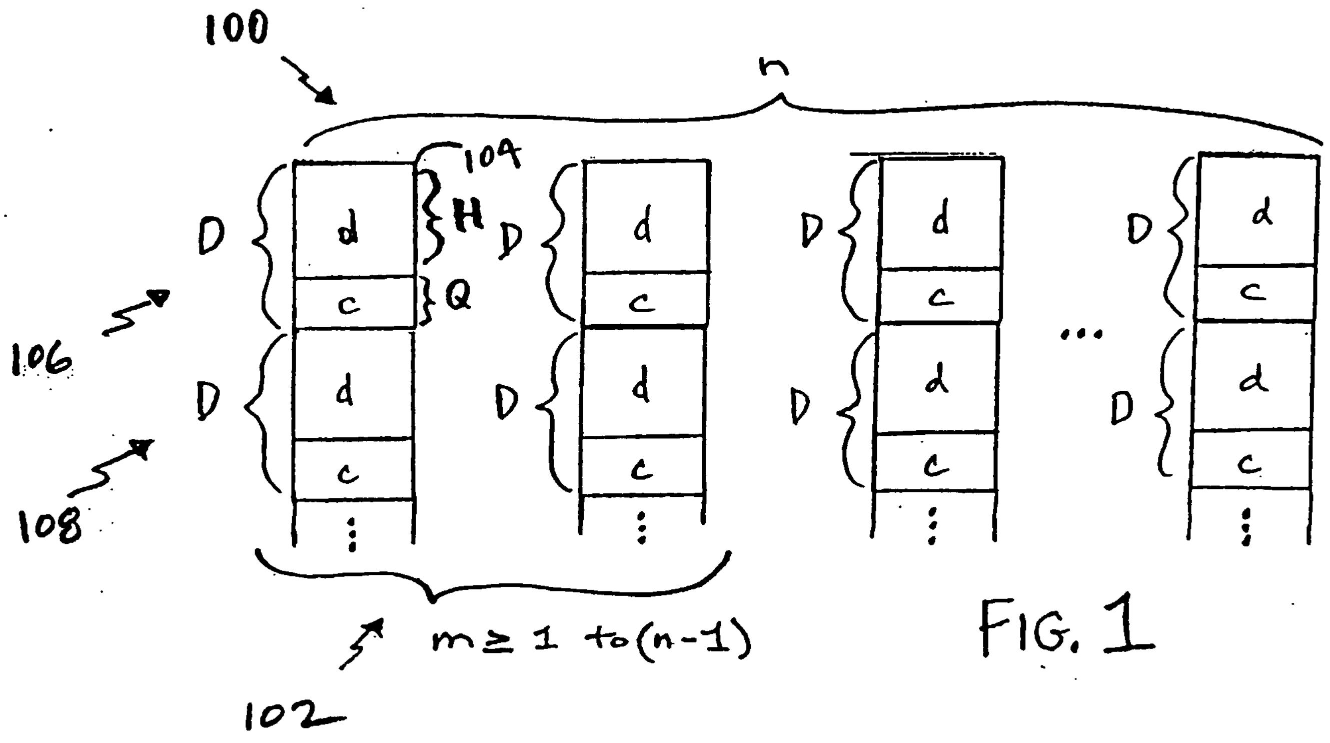


FIG. 1

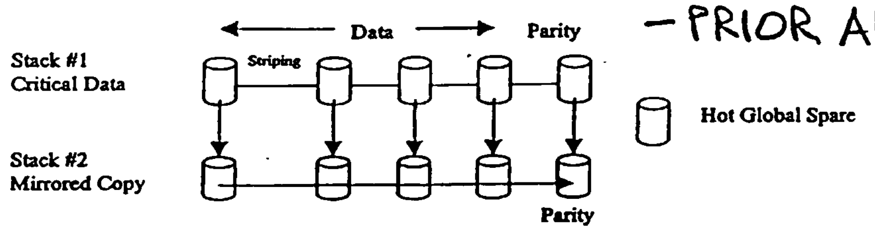


FIG. 2

-PRIOR ART-

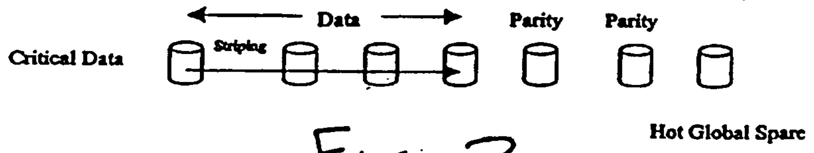
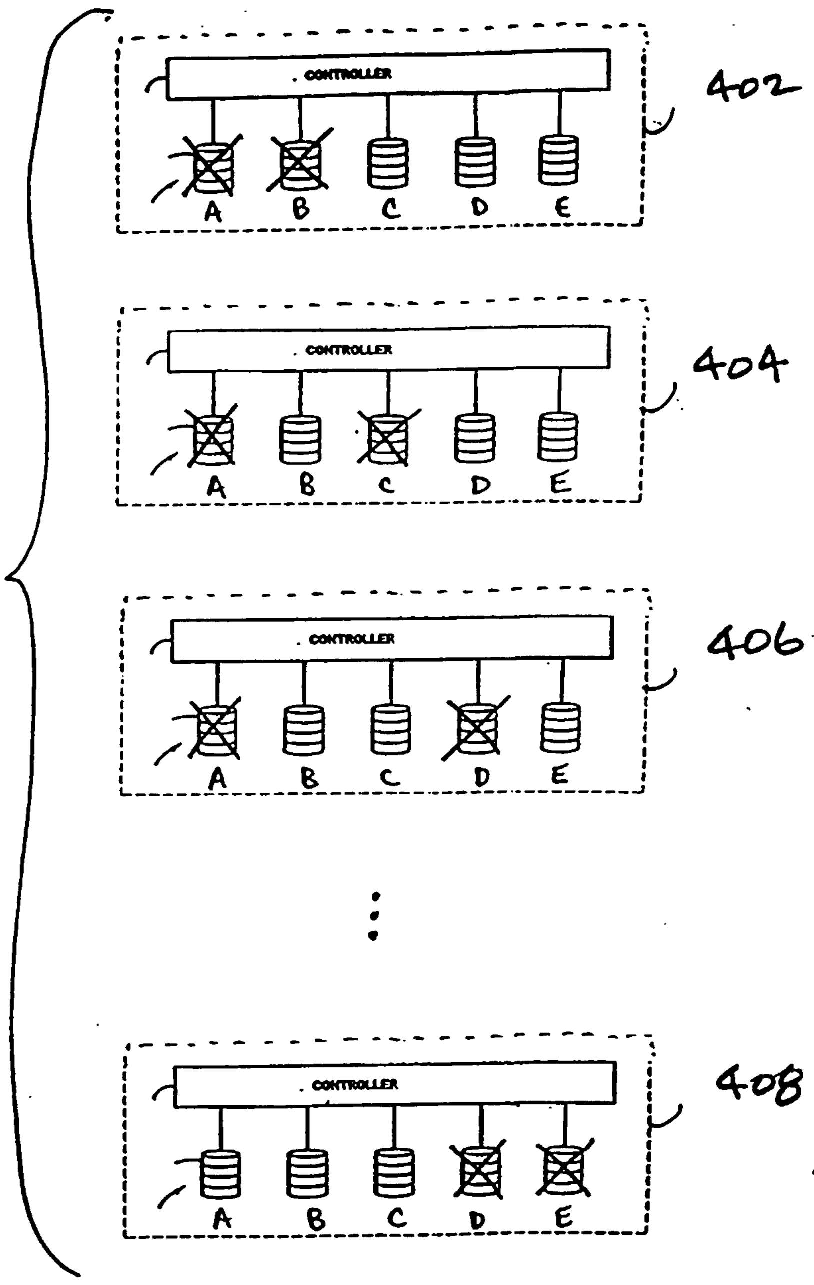


FIG. 3

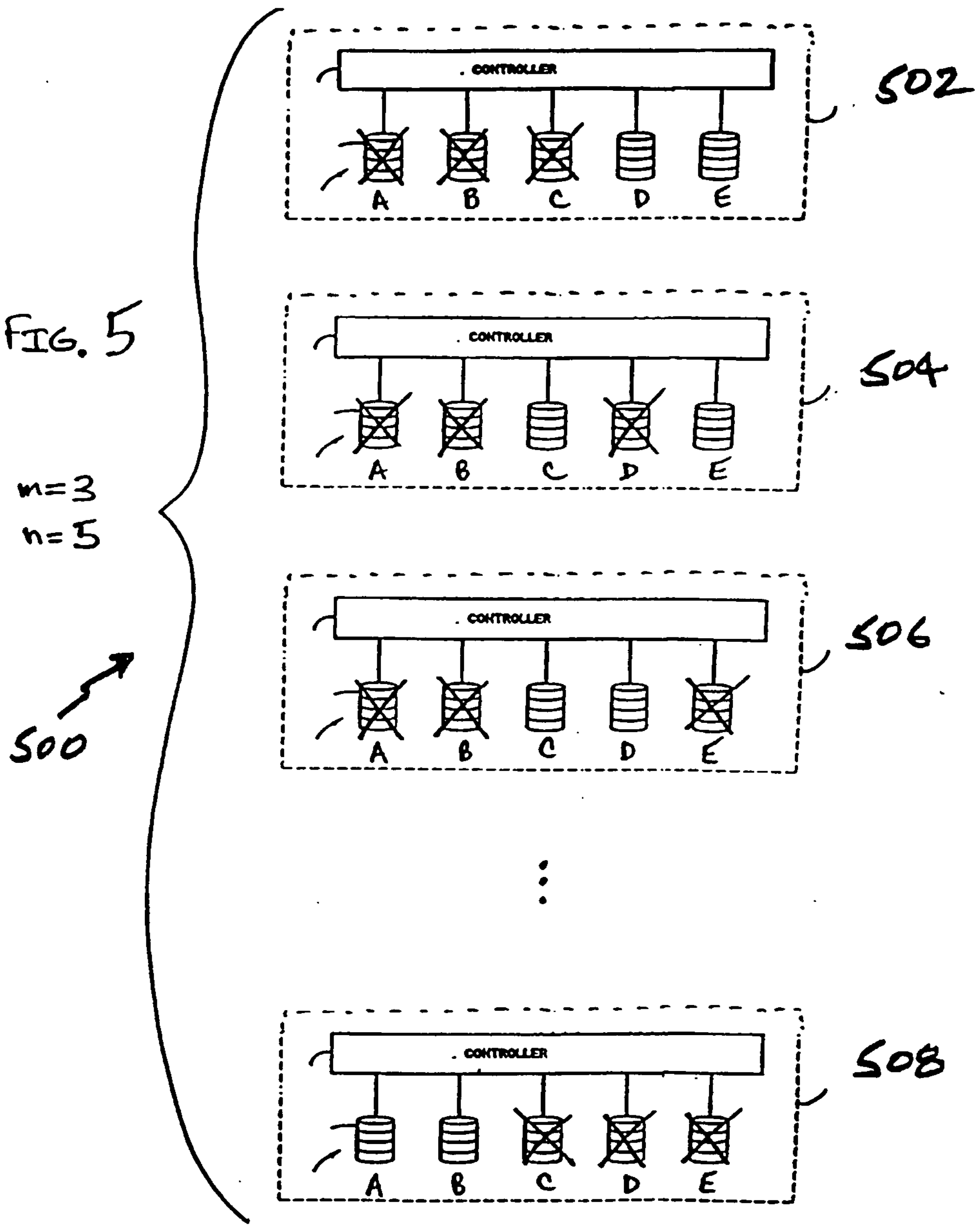


410

A	B	C	D	E	RECOVERABLE?
X	X	OK	OK	OK	YES
X	OK	X	OK	OK	YES
X	OK	OK	X	OK	YES
X	OK	OK	OK	X	YES
OK	X	X	OK	OK	YES
OK	X	OK	X	OK	YES
OK	X	OK	OK	X	YES
OK	OK	X	X	OK	YES
OK	OK	X	OK	X	YES
OK	OK	OK	X	X	YES

OK	-	data available
X	-	data unavailable

FIG. 5



510

A	B	C	D	E	RECOVERABLE?
X	X	X	OK	OK	YES
X	X	OK	X	OK	YES
X	X	OK	OK	X	YES
X	OK	X	X	OK	YES
X	OK	X	OK	X	YES
X	OK	OK	X	X	YES
OK	X	X	X	OK	YES
OK	X	X	OK	X	YES
OK	X	OK	X	X	YES
OK	OK	X	X	X	YES

OK	-	data available
X	-	data unavailable

FIG. 6

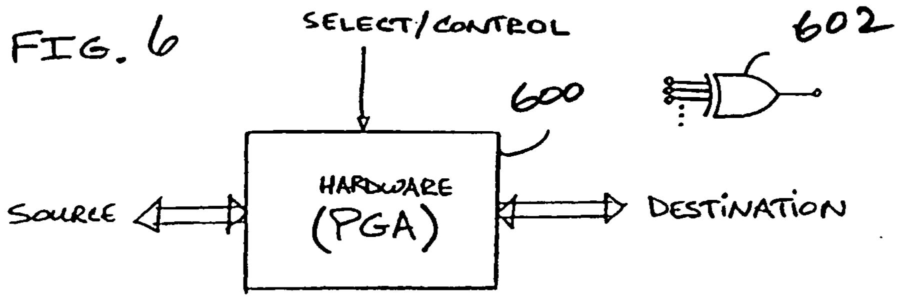
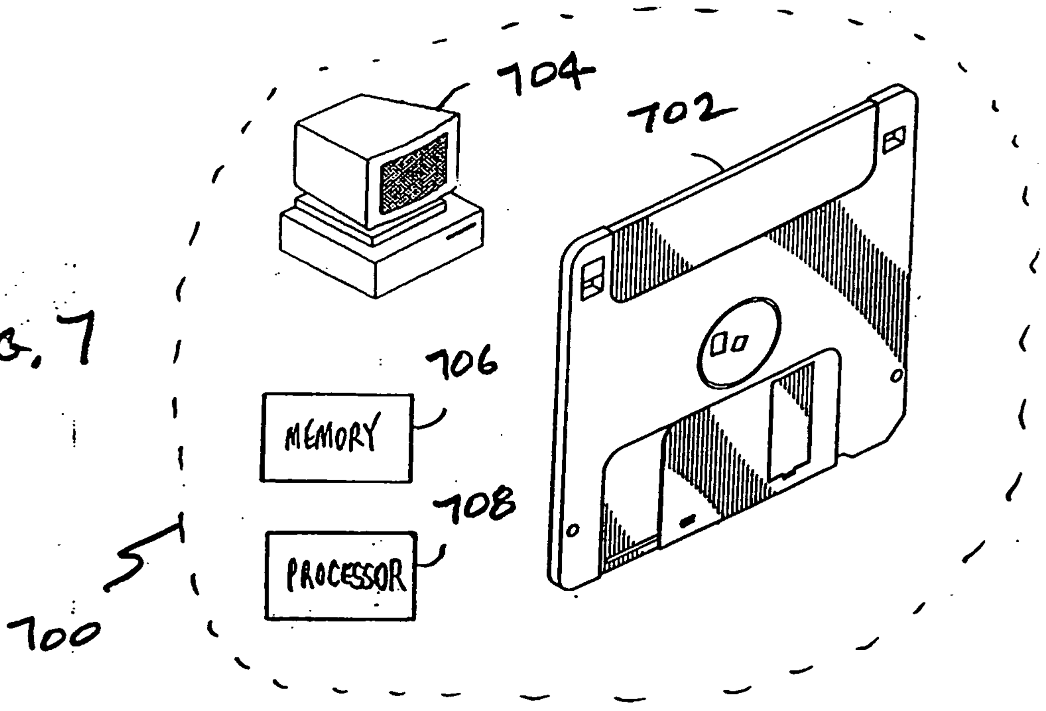
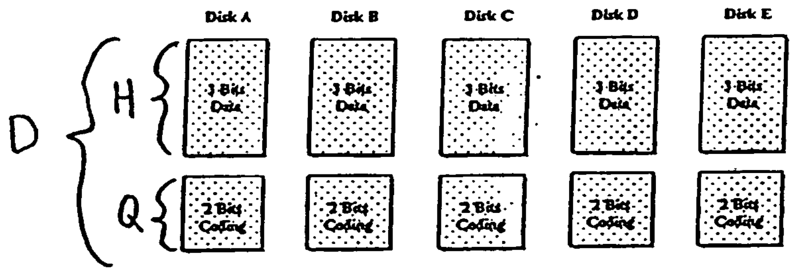

SELECT/CONTROL
602
600
HARDWARE
(PGA)
SOURCE
DESTINATION

FIG. 7


704
702
706
MEMORY
708
PROCESSOR
700

STRUCTURE AND SIZE
OF DATA
FOR
$m=2, n=5$



n = 5 disks total, m = 2 disks missing, p = 1,
Resulting in d = 3 data bits per disk and c = 2 coding bits per disk (15 data bits to the Array)

FIG. 8

REDUNDANCY DATA GENERATION FUNCTION MATRIX
FOR $m=2, n=5$

	A1	A2	A3	B1	B2	B3	C1	C2	C3	D1	D2	D3	E1	E2	E3
A'								•		•				•	
A"									•			•	•		
B'		•									•		•		
B"	•											•			•
C'	•				•									•	
C"			•	•											•
D'		•		•				•							
D"			•			•	•								
E'					•		•				•				
E"						•			•	•					

A' = C2 (+) D1 (+) E2
A" = C3 (+) D3 (+) E1

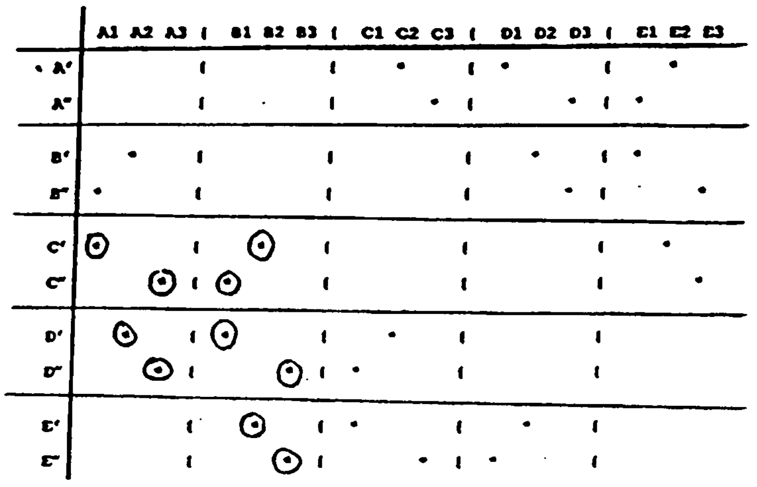
B' = A2 (+) D2 (+) E1
B" = A1 (+) D3 (+) E3

C' = A1 (+) B2 (+) E2
C" = A3 (+) B1 (+) E3

D' = A2 (+) B1 (+) C2
D" = A3 (+) B3 (+) C1

E' = B2 (+) C1 (+) D2
E" = B3 (+) C3 (+) D1

FIG. 9



A' = C2 (+) D1 (+) E2
A" = C3 (+) D3 (+) E1

B' = A2 (+) D2 (+) E1
B" = A1 (+) D3 (+) E3

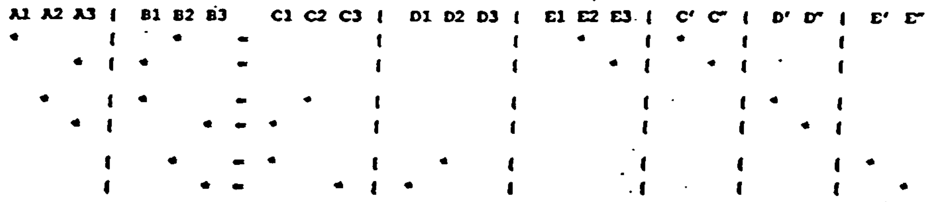
C' = A1 (+) B2 (+) E2
C" = A3 (+) B1 (+) E3

D' = A2 (+) B1 (+) C2
D" = A3 (+) B3 (+) C1

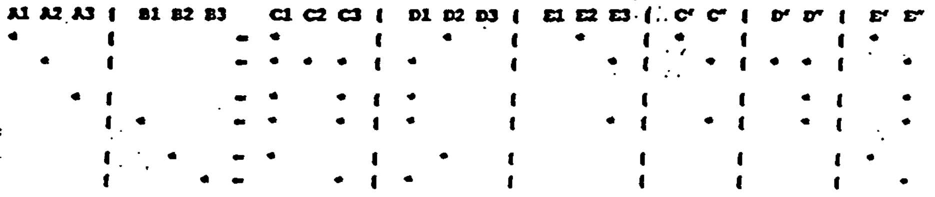
E' = B2 (+) C1 (+) D2
E" = B3 (+) C3 (+) D1

REDUNDANCY DATA GENERATION FUNCTION MATRIX
FOR m=2, n=5

Decoding Function Matrix for Disks A and B Missing
Surviving: C1, C2, C3, D1, D2, D3, E1, E2, E3, C', C", D', D", E', E"


A1 A2 A3 B1 B2 B3 C1 C2 C3 D1 D2 D3 E1 E2 E3 C' C" D' D" E' E"

Solved Decoding Function Matrix for Disks A and B Missing

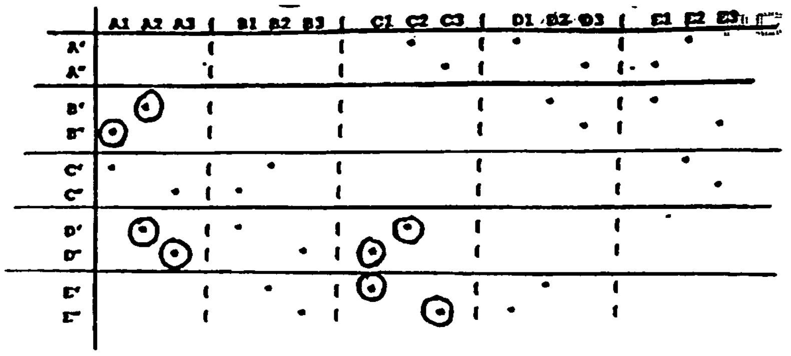

A1 A2 A3 B1 B2 B3 C1 C2 C3 D1 D2 D3 E1 E2 E3 C' C" D' D" E' E"

Complete Decoding Functions for Disks A and B Missing

A1 = C1 (+) D2 (+) E2 (+) C' (+) E'
A2 = C1 (+) C2 (+) C3 (+) D1 (+) E3 (+) C" (+) D' (+) D" (+) E"
A3 = C1 (+) C3 (+) D1 (+) D" (+) E"

B1 = C1 (+) C3 (+) D1 (+) E3 (+) C" (+) D" (+) E"
B2 = C1 (+) D2 (+) E'
B3 = C3 (+) D1 (+) E"

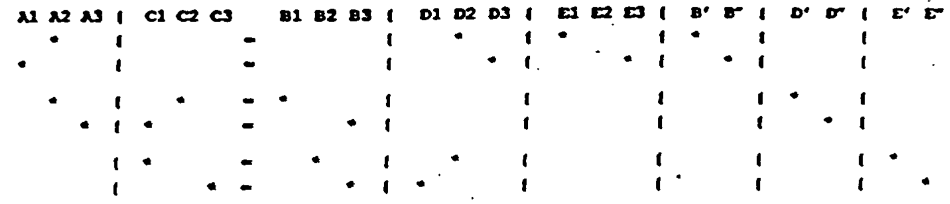
FIG. 10



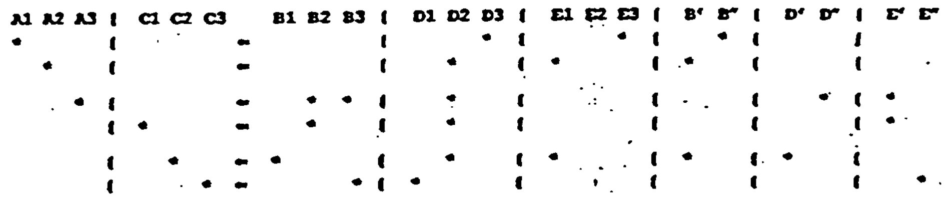
A' = C2 (+) D1 (+) E2
A" = C3 (+) D3 (+) E1
B' = A2 (+) D2 (+) E1
B" = A1 (+) D3 (+) E3
C' = A1 (+) B2 (+) E2
C" = A3 (+) B1 (+) E3
D' = A2 (+) B1 (+) C2
D" = A3 (+) B3 (+) C1
E' = B2 (+) C1 (+) D2
E" = B3 (+) C3 (+) D1

REDUNDANCY DATA GENERATION FUNCTION MATRIX
FOR m=2, n=5

Decoding Function Matrix for Disks A and C Missing
Surviving: B1, B2, B3, D1, D2, D3, E1, E2, E3, B', B", D', D", E', E"


A1 A2 A3 C1 C2 C3 B1 B2 B3 D1 D2 D3 E1 E2 E3 B' B" D' D" E' E"

Solved Decoding Function Matrix for Disks A and C Missing

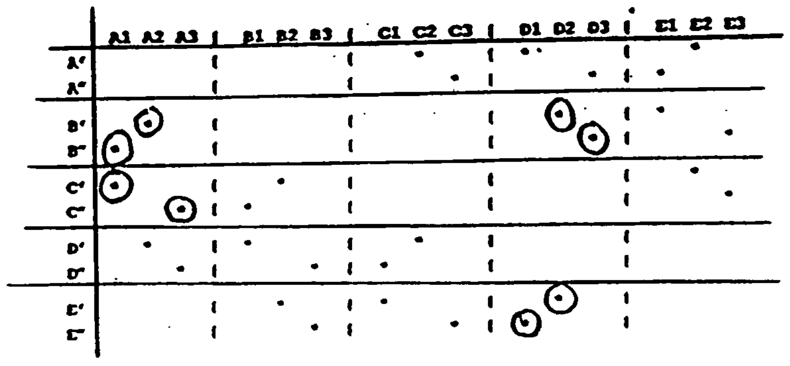

A1 A2 A3 C1 C2 C3 B1 B2 B3 D1 D2 D3 E1 E2 E3 B' B" D' D" E' E"

Complete Decoding Functions for Disks A and C Missing

A1 = D3 (+) E3 (+) B"
A2 = D2 (+) E1 (+) B'
A3 = B2 (+) B3 (+) D2 (+) D" (+) E'

C1 = B2 (+) D2 (+) E'
C2 = B1 (+) D2 (+) E1 (+) B' (+) D'
C3 = B3 (+) D1 (+) E"

FIG. 11



A' = C2 (+) D1 (+) E2
A" = C3 (+) D3 (+) E1

B' = A2 (+) D2 (+) E1
B" = A1 (+) D3 (+) E3

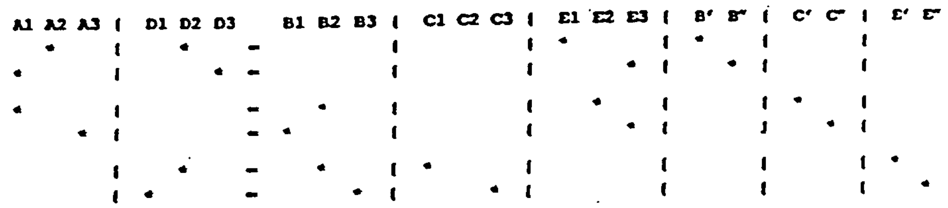
C' = A1 (+) B2 (+) E2
C" = A3 (+) B1 (+) E3

D' = A2 (+) B1 (+) C2
D" = A3 (+) B3 (+) C1

E' = B2 (+) C1 (+) D2
E" = B3 (+) C3 (+) D1

REDUNDANCY DATA GENERATION FUNCTION MATRIX
FOR m=2, n=5

Decoding Function Matrix for Disks A and D Missing
Surviving: B1, B2, B3, C1, C2, C3, E1, E2, E3, B', B", C', C", E', E"



Solved Decoding Function Matrix for Disks A and D Missing

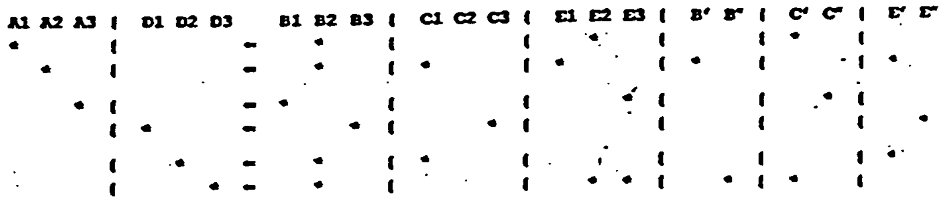
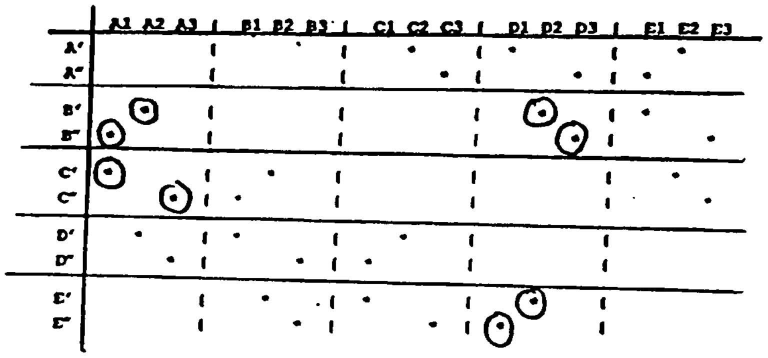


FIG. 12

Complete Decoding Functions for Disks A and D Missing

A1 = B2 (+) E2 (+) C'
A2 = B2 (+) C1 (+) E1 (+) B' (+) E'
A3 = B1 (+) E3 (+) C"

D1 = B3 (+) C3 (+) E"
D2 = B2 (+) C1 (+) E'
D3 = B2 (+) E2 (+) E3 (+) B" (+) C'


A1 A2 A3 B1 B2 B3 C1 C2 C3 D1 D2 D3 E1 E2 E3
A' A" B' B" C' C" D' D" E' E"

A' = C2 (+) D1 (+) E2
A" = C3 (+) D3 (+) E1

B' = A2 (+) D2 (+) E1
B" = A1 (+) D3 (+) E3

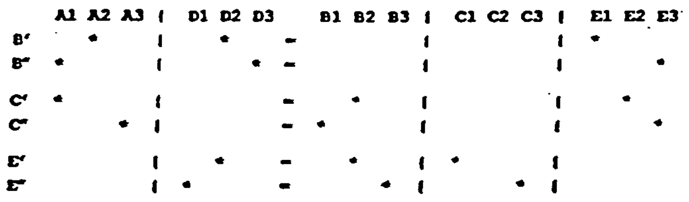
C' = A1 (+) B2 (+) E2
C" = A3 (+) B1 (+) E3

D' = A2 (+) B1 (+) C2
D" = A3 (+) B3 (+) C1

E' = B2 (+) C1 (+) D2
E" = B3 (+) C3 (+) D1

REDUNDANCY DATA GENERATION FUNCTION MATRIX
FOR m=2, n=5

Decoding Function Matrix for Disks A and D Missing
Surviving: B1, B2, B3, C1, C2, C3, E1, E2, E3, B', B", C', C", E', E"



Rearrangement of Decoding Function Matrix for Disks A and D Missing

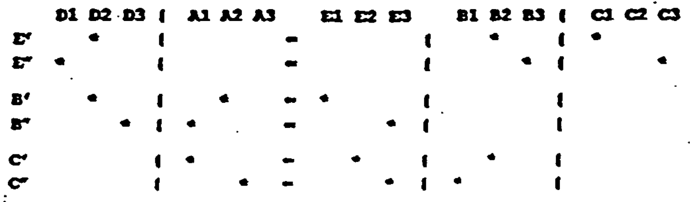


FIG. 13

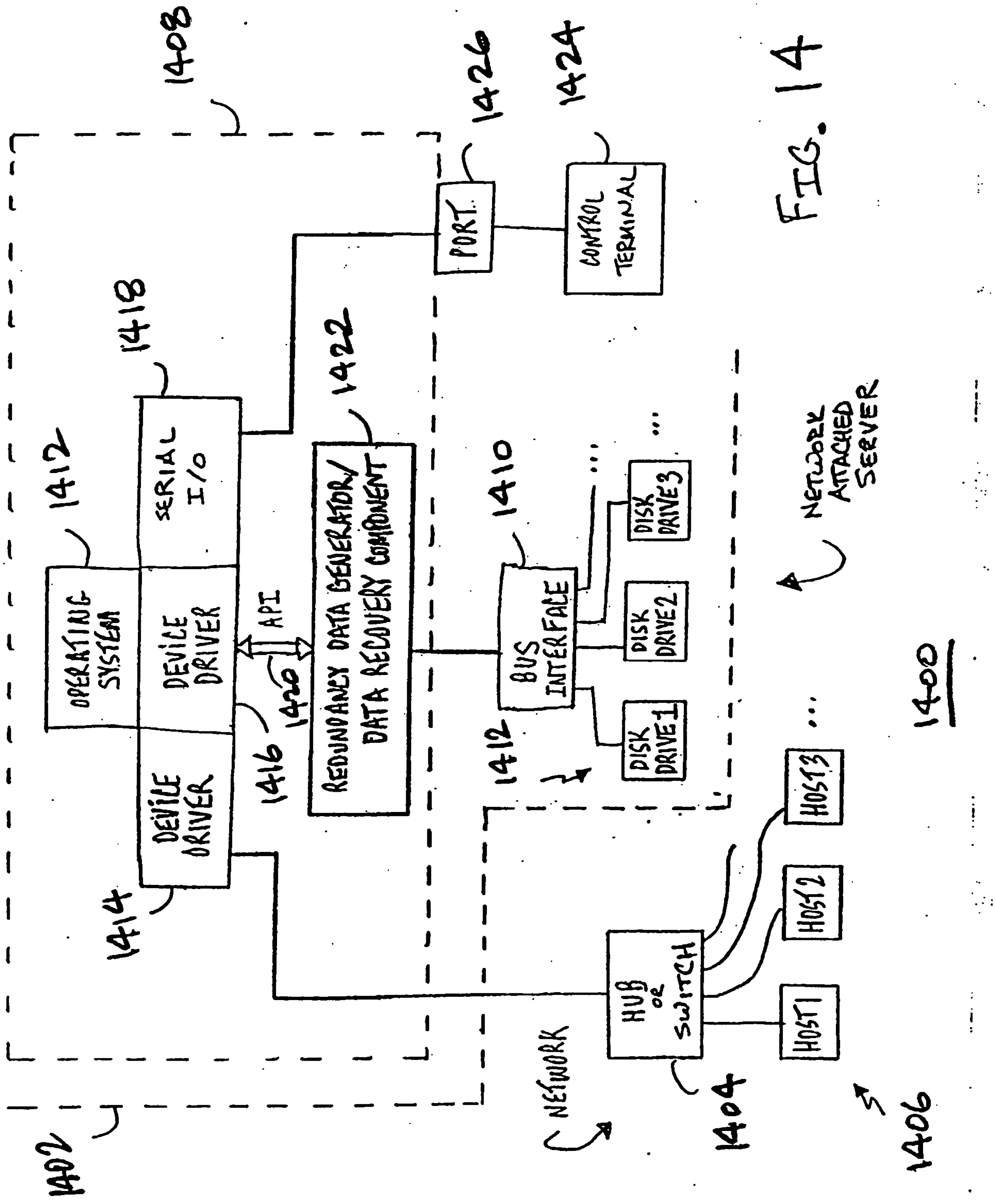

1408
1426
1424
PORT
CONTROL TERMINAL
1418
1412
1422
OPERATING SYSTEM
SERIAL I/O
DEVICE DRIVER
API
1420
REDUNDANCY DATA GENERATOR/ DATA RECOVERY COMPONENT
1410
BUS INTERFACE
DISK DRIVE1
DISK DRIVE2
DISK DRIVE3
NETWORK ATTACHED SERVER
1416
1412
DEVICE DRIVER
1414
HUB or SWITCH
HOST1
HOST2
HOST3
NETWORK
1404
1406
1400
1402

FIG. 14

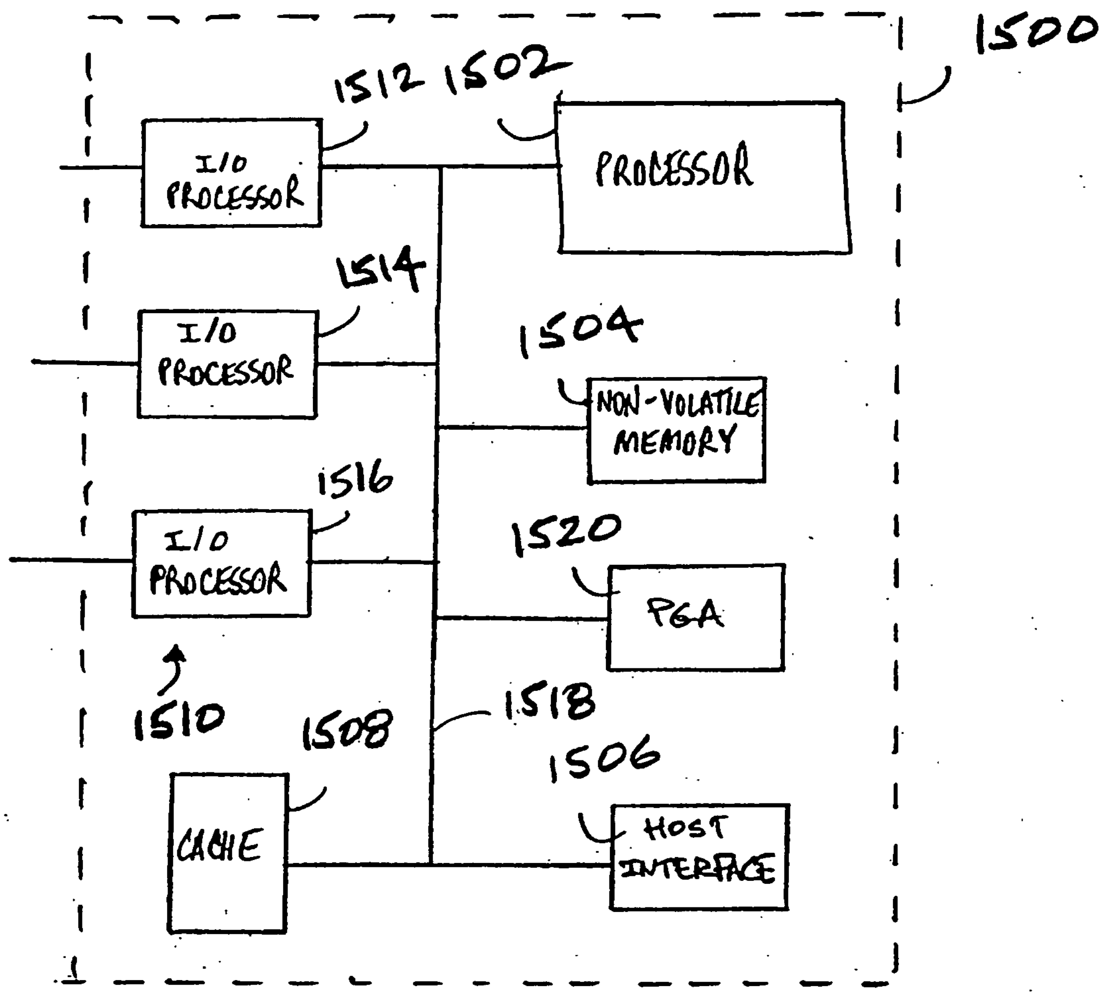

1500
1512
1502
I/O PROCESSOR
PROCESSOR
1514
I/O PROCESSOR
1504
NON-VOLATILE MEMORY
1516
I/O PROCESSOR
1520
PGA
1510
1508
1518
1506
CACHE
HOST INTERFACE

FIG. 15

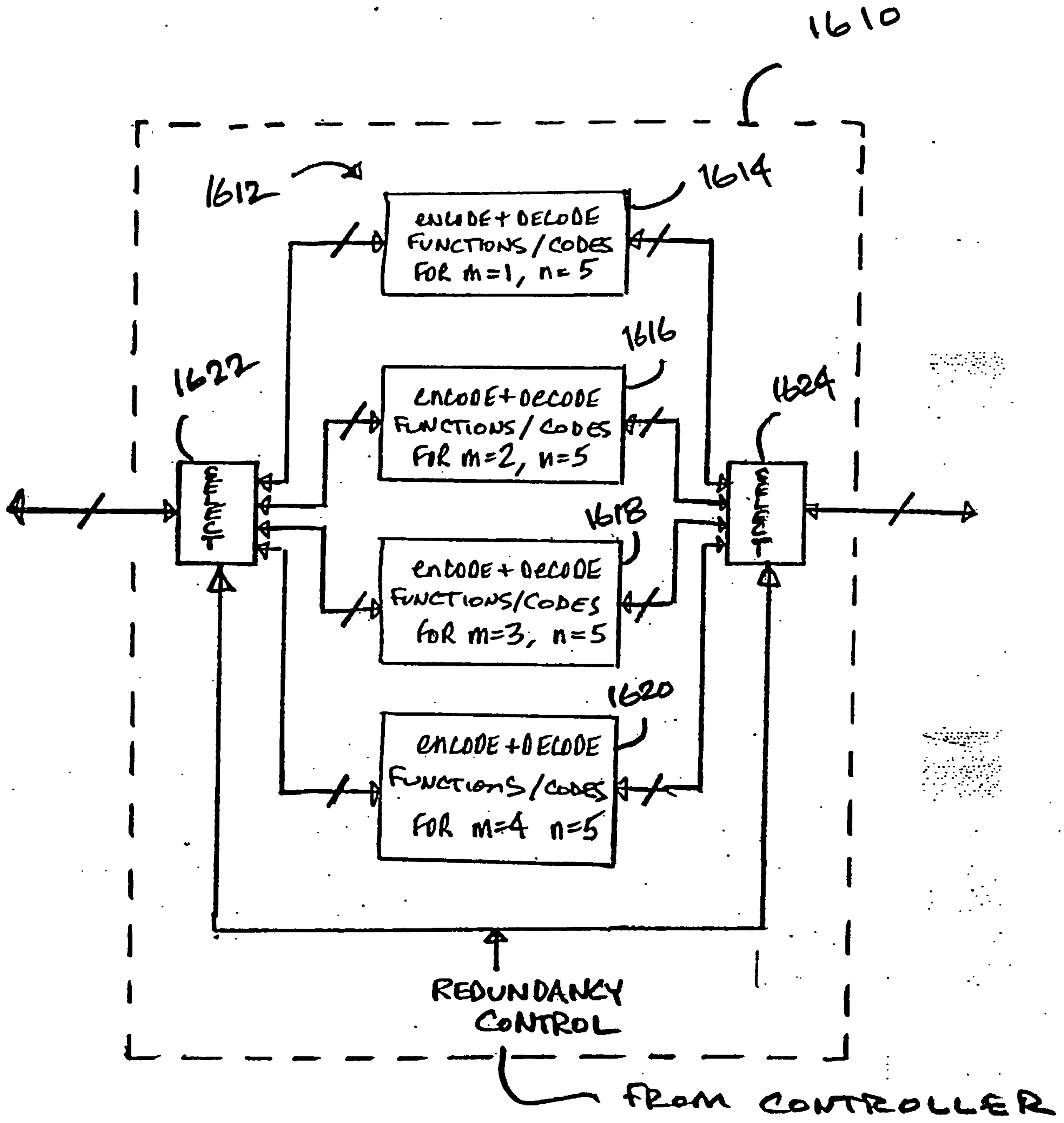

1610
1612
1614
ENCODE + DECODE FUNCTIONS / CODES FOR m=1, n=5
1616
ENCODE + DECODE FUNCTIONS / CODES FOR m=2, n=5
1618
ENCODE + DECODE FUNCTIONS / CODES FOR m=3, n=5
1620
ENCODE + DECODE FUNCTIONS / CODES FOR m=4 n=5
1622
SELECT
1624
SELECT
REDUNDANCY CONTROL
FROM CONTROLLER

FIG. 16.1

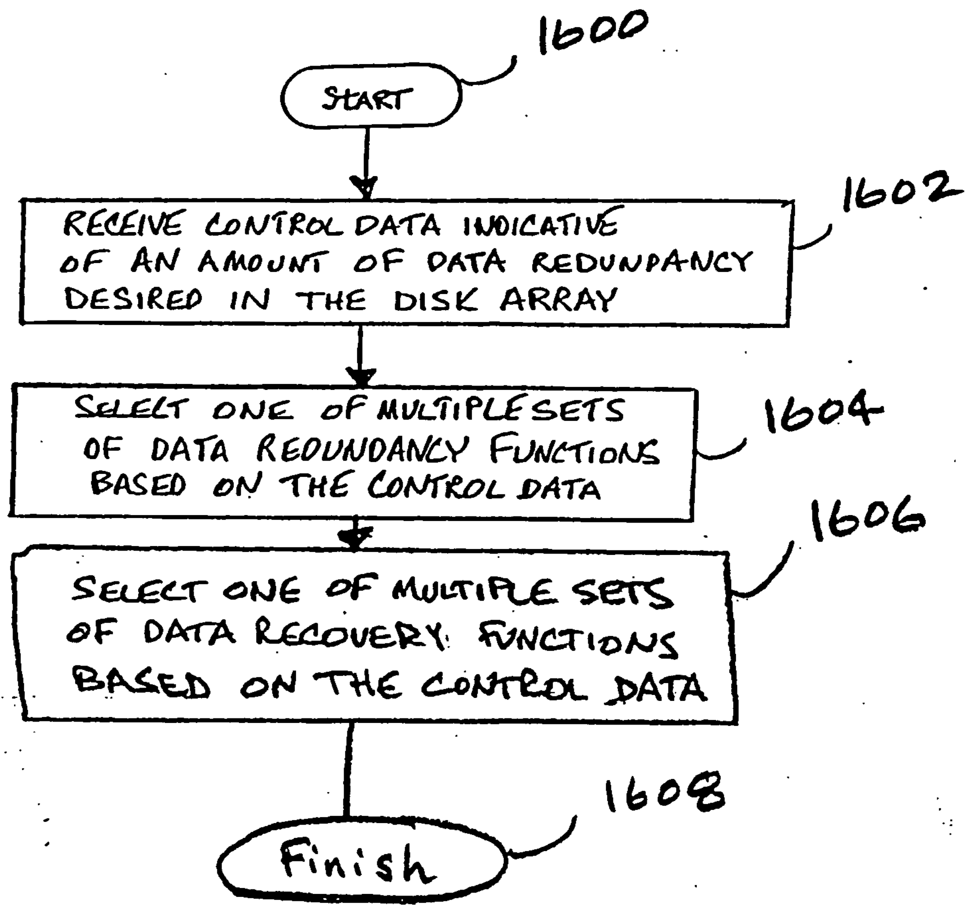

1600
START
1602
RECEIVE CONTROL DATA INDICATIVE OF AN AMOUNT OF DATA REDUNDANCY DESIRED IN THE DISK ARRAY
1604
SELECT ONE OF MULTIPLE SETS OF DATA REDUNDANCY FUNCTIONS BASED ON THE CONTROL DATA
1606
SELECT ONE OF MULTIPLE SETS OF DATA RECOVERY FUNCTIONS BASED ON THE CONTROL DATA
1608
Finish

FIG. 16.2

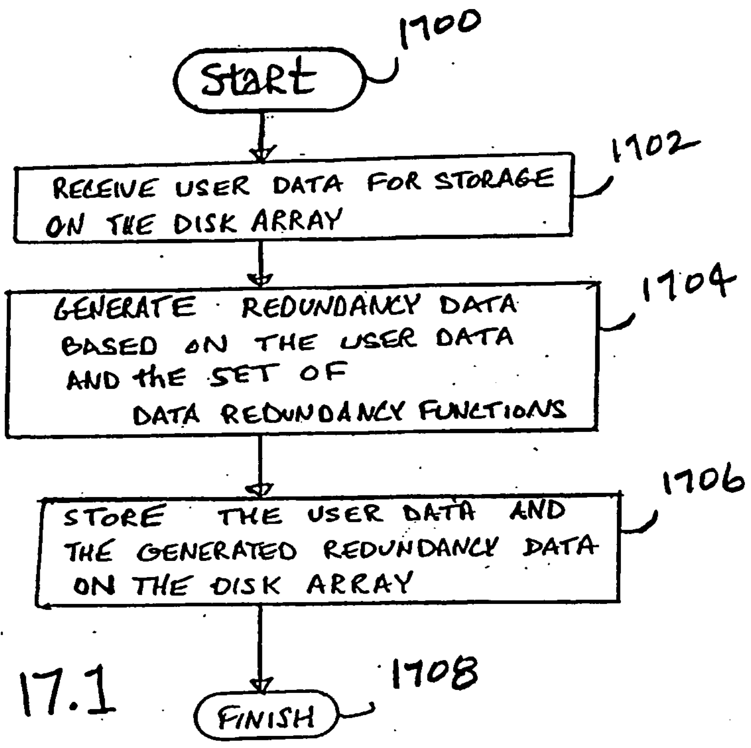

1700
Start
1702
RECEIVE USER DATA FOR STORAGE ON THE DISK ARRAY
1704
GENERATE REDUNDANCY DATA BASED ON THE USER DATA AND the SET OF DATA REDUNDANCY FUNCTIONS
1706
STORE THE USER DATA AND THE GENERATED REDUNDANCY DATA ON THE DISK ARRAY
1708
FINISH

FIG. 17.1

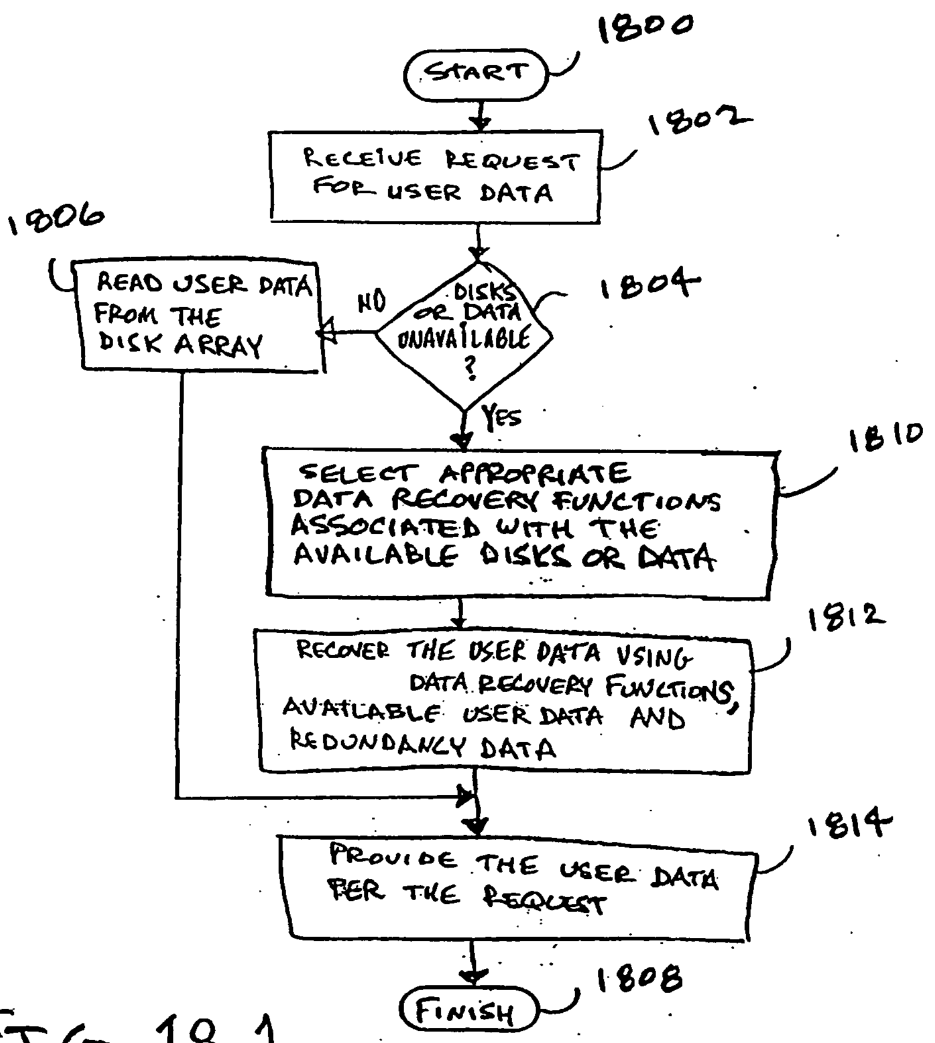

1800
START
1802
RECEIVE REQUEST FOR USER DATA
1806
READ USER DATA FROM THE DISK ARRAY
NO
DISKS OR DATA UNAVAILABLE ?
1804
YES
1810
SELECT APPROPRIATE DATA RECOVERY FUNCTIONS ASSOCIATED WITH THE AVAILABLE DISKS OR DATA
1812
RECOVER THE USER DATA USING DATA RECOVERY FUNCTIONS, AVAILABLE USER DATA AND REDUNDANCY DATA
1814
PROVIDE THE USER DATA PER THE REQUEST
1808
FINISH

FIG. 18.1

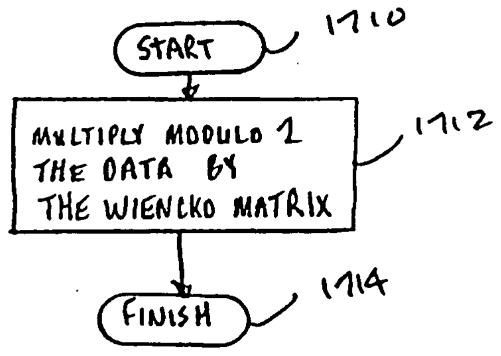


FIG. 17.2

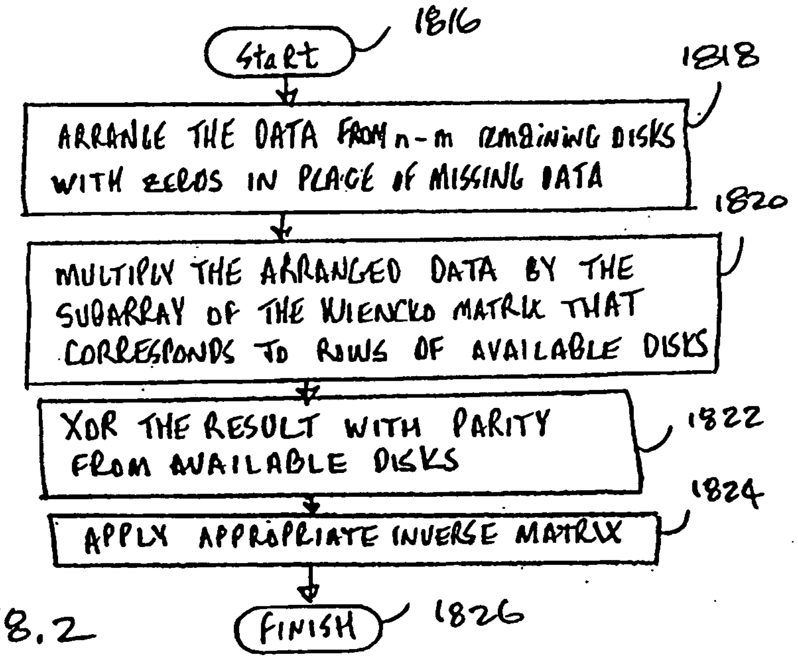

Start
1816
ARRANGE THE DATA FROM n-m REMAINING DISKS WITH ZEROS IN PLACE OF MISSING DATA
1818
MULTIPLY THE ARRANGED DATA BY THE SUBARRAY OF THE WIENCKO MATRIX THAT CORRESPONDS TO ROWS OF AVAILABLE DISKS
1820
XOR THE RESULT WITH PARITY FROM AVAILABLE DISKS
1822
APPLY APPROPRIATE INVERSE MATRIX
1824
FINISH
1826

FIG. 18.2

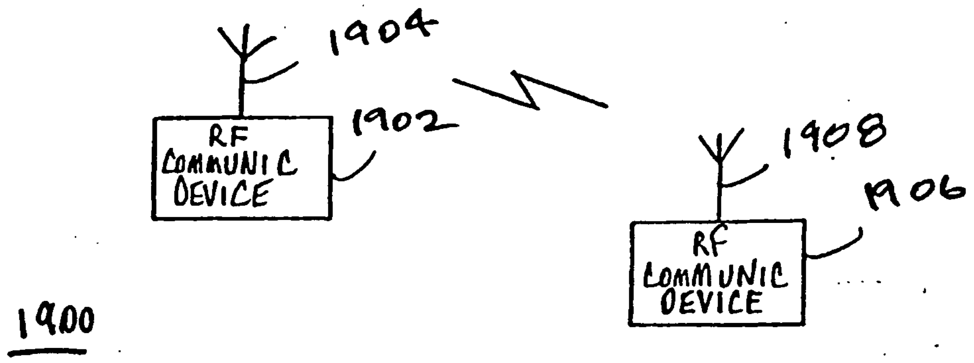

1904
RF COMMUNIC DEVICE
1902
1908
1906
RF COMMUNIC DEVICE
1900

FIG. 19

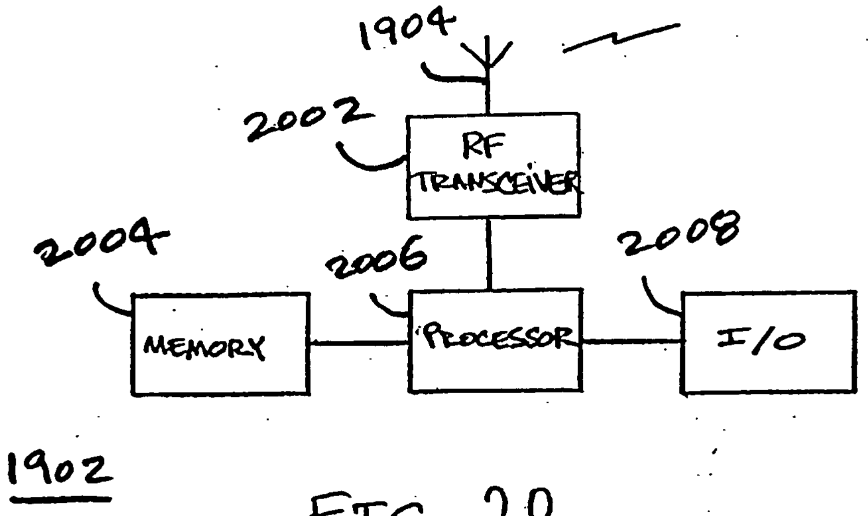

1904
2002
RF TRANSCEIVER
2004
2006
2008
MEMORY
PROCESSOR
I/O
1902

FIG. 20

FIG. 21

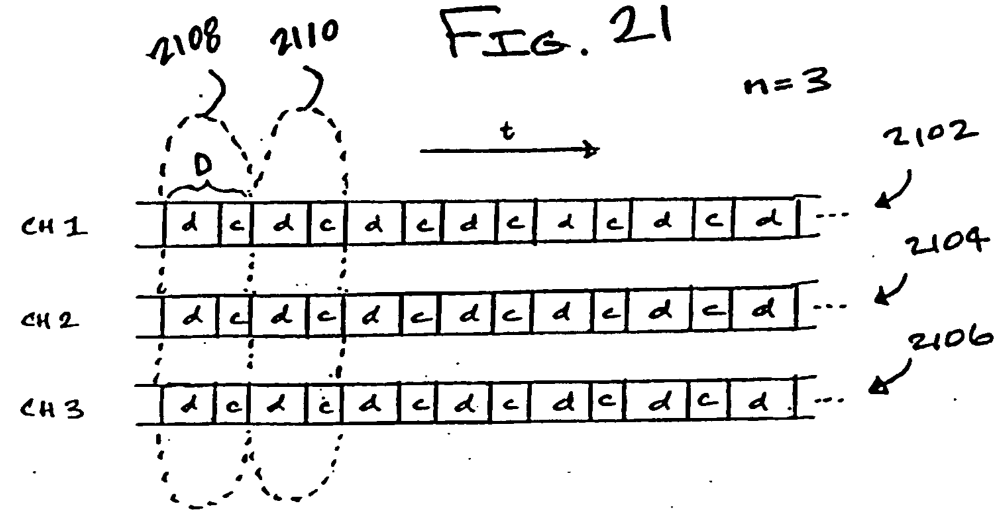

2108
2110
n=3
t
D
2102
2104
2106
CH 1
CH 2
CH 3
d
c

FIG. 22

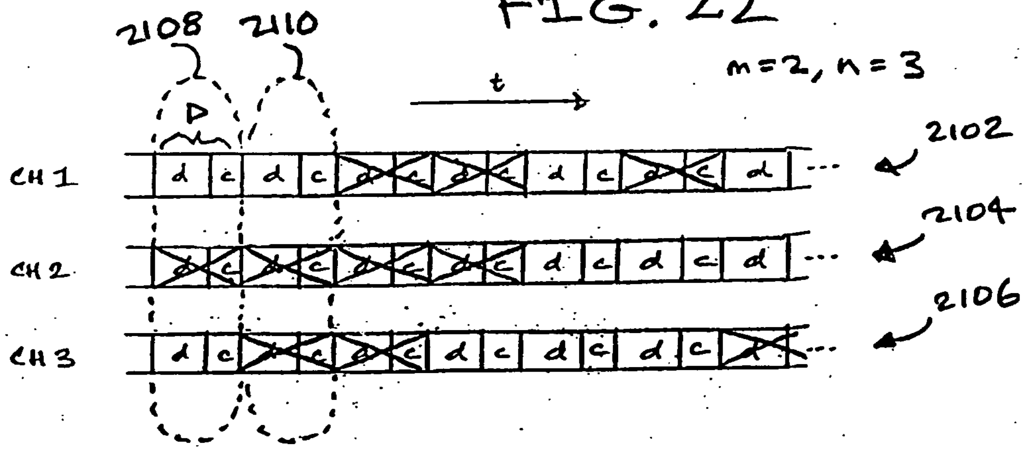
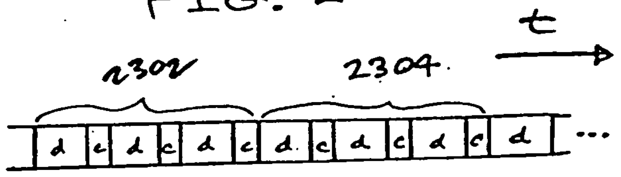
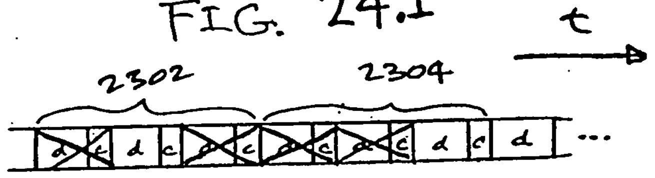

2108
2110
m=2, n=3
t
D
2102
2104
2106
CH 1
CH 2
CH 3
d
c

FIG. 23



n=3

FIG. 24.1



m=2, n=3 DATA RECOVERABLE

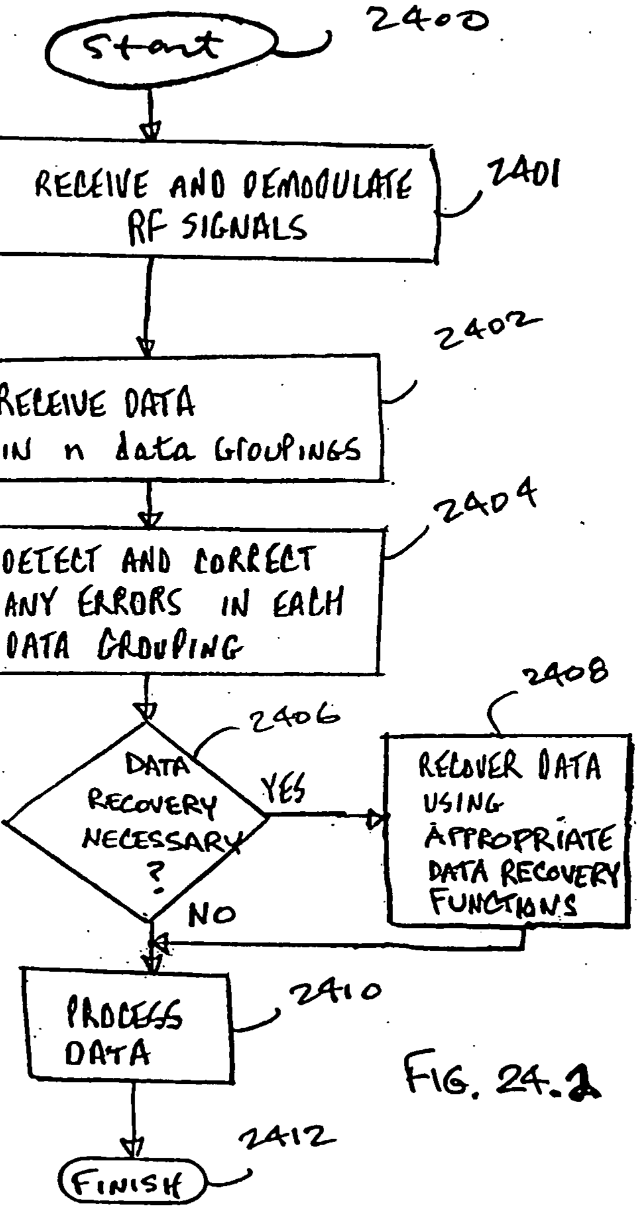

Start
2400
RECEIVE AND DEMODULATE RF SIGNALS
2401
RECEIVE DATA IN n data GROUPINGS
2402
DETECT AND CORRECT ANY ERRORS IN EACH DATA GROUPING
2404
DATA RECOVERY NECESSARY ?
2406
YES
NO
RECOVER DATA USING APPROPRIATE DATA RECOVERY FUNCTIONS
2408
PROCESS DATA
2410
FINISH
2412

FIG. 24.2

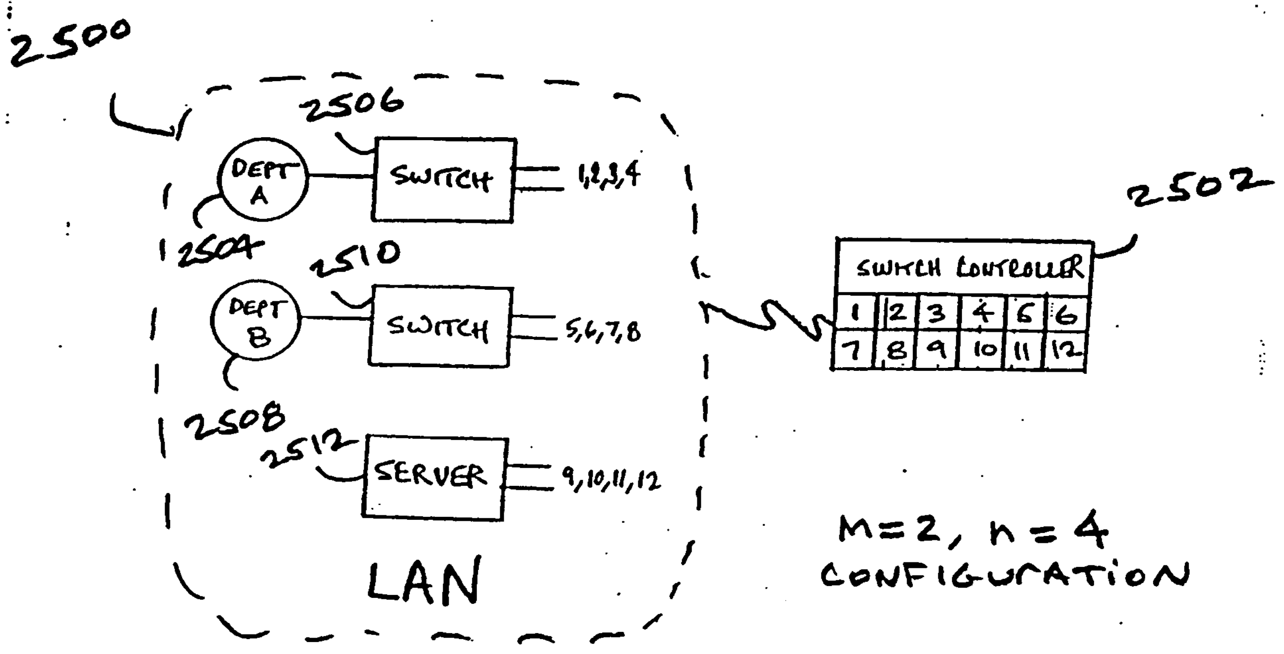

2500
2506
DEPT A
SWITCH
1,2,3,4
2504
2510
DEPT B
SWITCH
5,6,7,8
2508
2512
SERVER
9,10,11,12
LAN
2502
SWITCH CONTROLLER
1 2 3 4 6 6
7 8 9 10 11 12
M=2, n=4
CONFIGURATION

FIG. 25

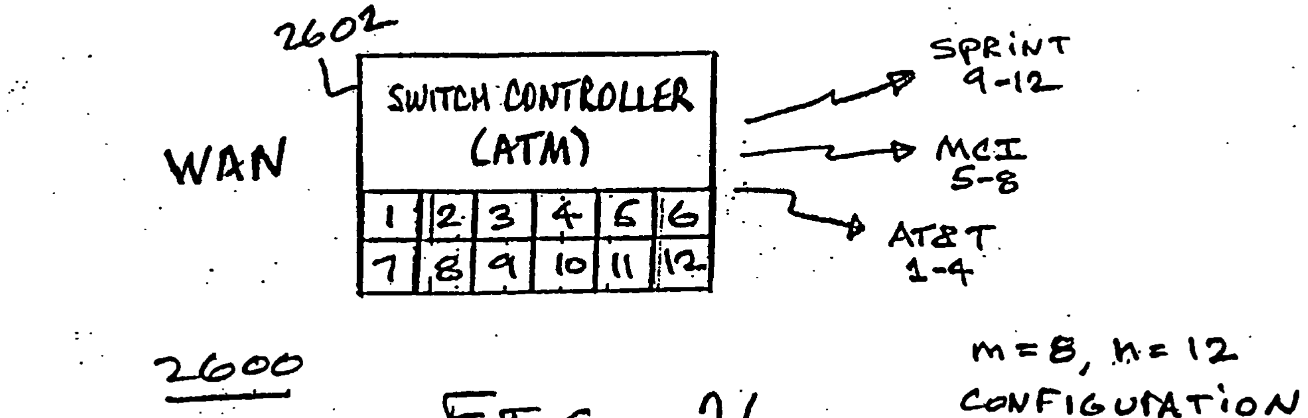
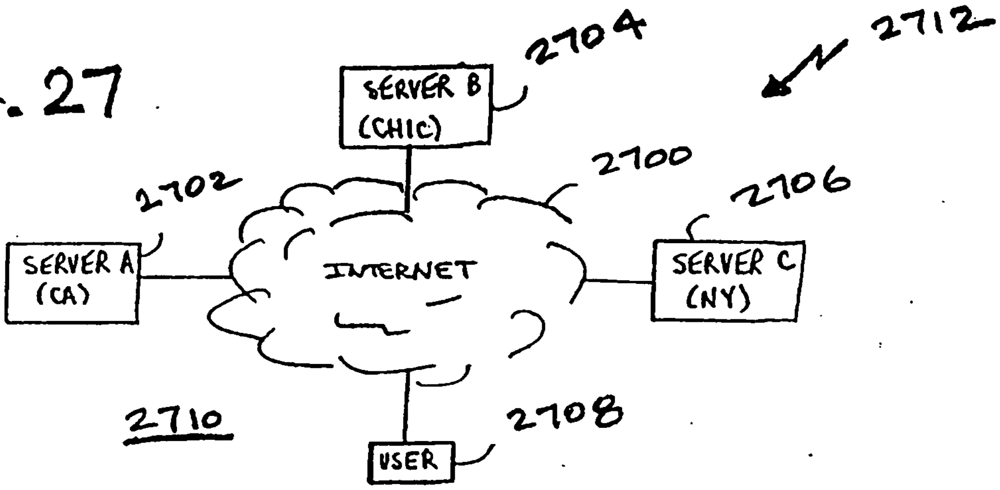

2602
SWITCH CONTROLLER (ATM)
1 2 3 4 6 6
7 8 9 10 11 12
WAN
SPRINT 9-12
MCI 5-8
AT&T 1-4
2600
m=8, n=12
CONFIGURATION

FIG. 26

FIG. 27



m=2, n=3

FIG. 28

SERVER	TIME DATA RCV'D	SELECTED SERVER?
A	10 MSEC	YES
B	25 MSEC	NO
C	30 MSEC	NO

Time = 7:30 a.m.

FIG. 29

SERVER	TIME DATA RCV'D	SELECTED SERVER?
A	50 MSEC	NO
B	25 MSEC	YES
C	30 MSEC	NO

Time = 8:30 a.m.

FIG. 30

SERVER	TIME DATA RCV'D	SELECTED SERVER?
A	50 MSEC	NO
B	25 + 50 = 75 MSEC	NO
C	30 MSEC	YES

Time = 11:30 a.m.

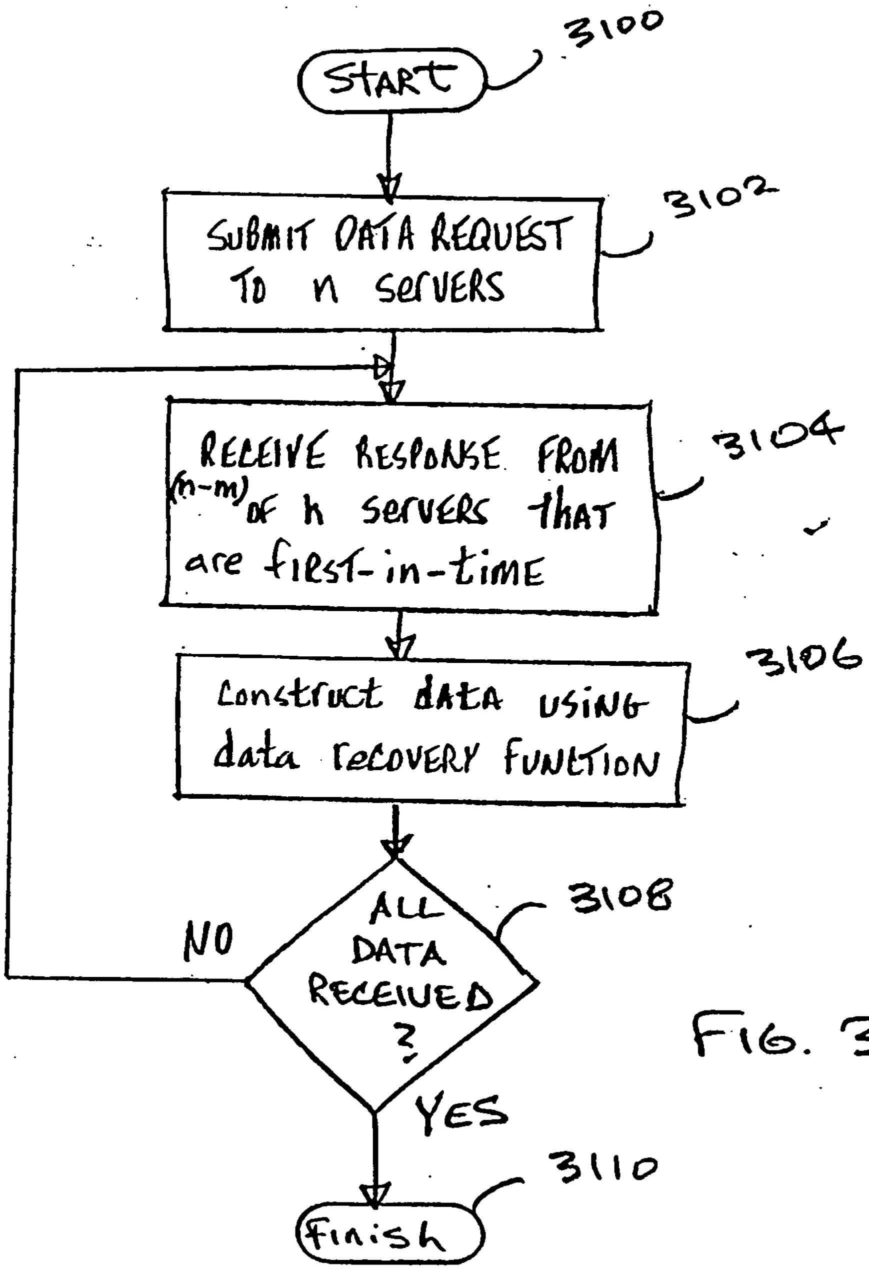

Start
3100
SUBMIT DATA REQUEST TO n SERVERS
3102
RECEIVE RESPONSE FROM (n-m) OF n SERVERS THAT are first-in-time
3104
Construct data using data recovery function
3106
ALL DATA RECEIVED ?
3108
NO
YES
Finish
3110

FIG. 31

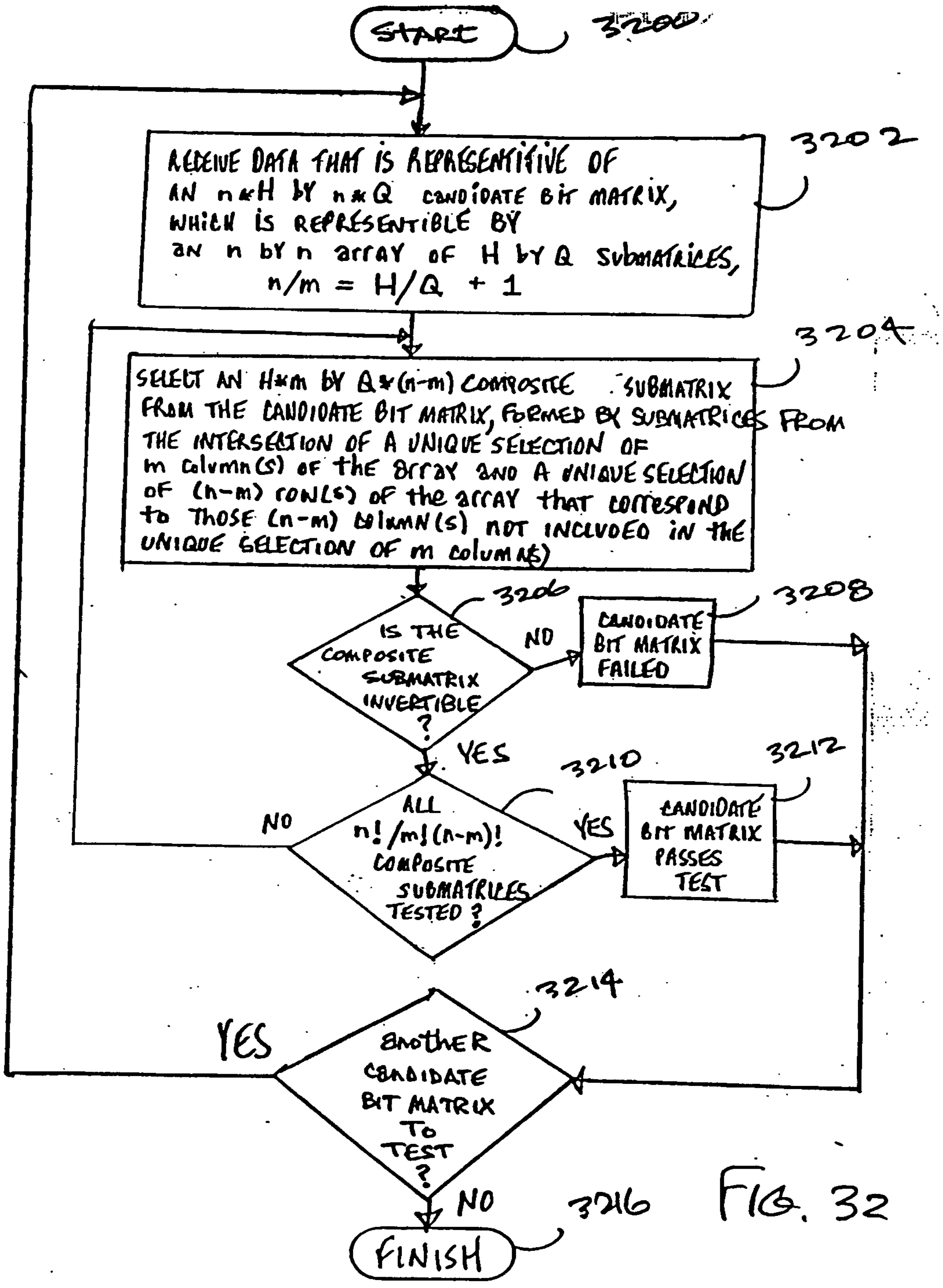
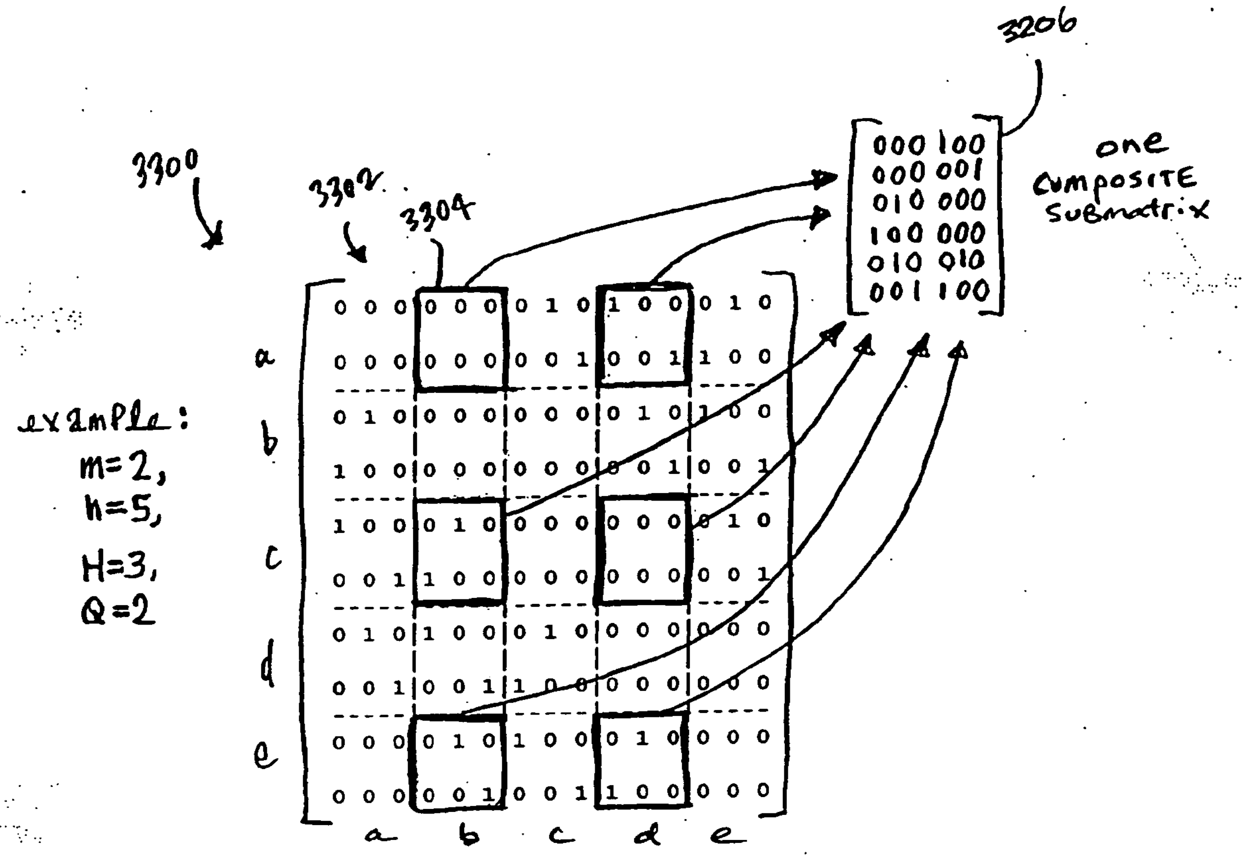

START
3200
3202
RECEIVE DATA THAT IS REPRESENTATIVE OF AN n*H BY n*Q CANDIDATE BIT MATRIX, WHICH IS REPRESENTIBLE BY AN n BY n ARRAY OF H BY Q SUBMATRICES, n/m = H/Q + 1
3204
SELECT AN H*m BY Q*(n-m) COMPOSITE SUBMATRIX FROM THE CANDIDATE BIT MATRIX, FORMED BY SUBMATRICES FROM THE INTERSECTION OF A UNIQUE SELECTION OF m column(s) of the array and a UNIQUE SELECTION of (n-m) row(s) of the array that correspond to those (n-m) column(s) not included in the UNIQUE SELECTION OF m columns)
3206
IS THE COMPOSITE SUBMATRIX INVERTIBLE ?
NO
CANDIDATE BIT MATRIX FAILED
3208
YES
3210
ALL n!/m!(n-m)! COMPOSITE SUBMATRICES TESTED ?
NO
YES
CANDIDATE BIT MATRIX PASSES TEST
3212
3214
YES
ANOTHER CANDIDATE BIT MATRIX TO TEST ?
NO
3216
FINISH

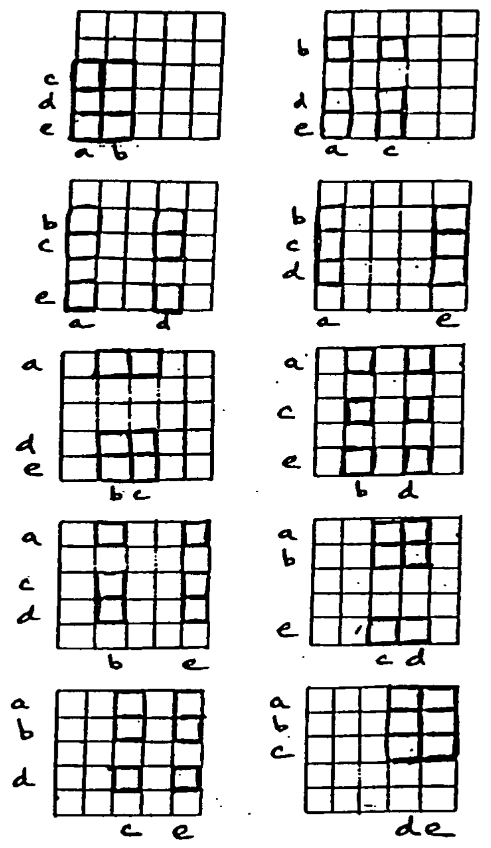
FIG. 32



n*H by n*Q bit matrix representible by an n by n array of H by Q bit submatrices where n/m = H/Q + 1

FIG. 33

example:
m=2
n=5



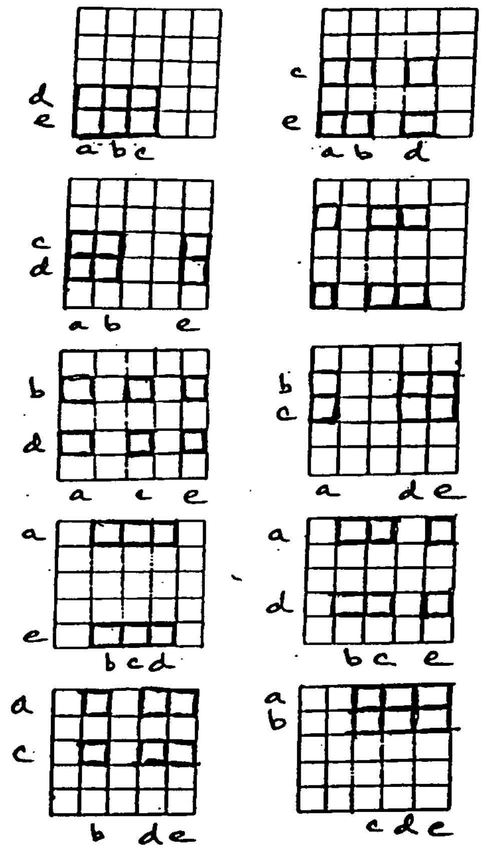
DEFINING SUBARRAYS AND COMPOSITE SUBMATRICES FOR $\begin{matrix} m=2 \\ n=5 \end{matrix}$

SUBARRAY DIMENSIONS = $m \times (n-m) = 2 \times 3$

OF COMPOSITE SUBMATRICES = $n!/m!(n-m)! = 10$

FIG. 34

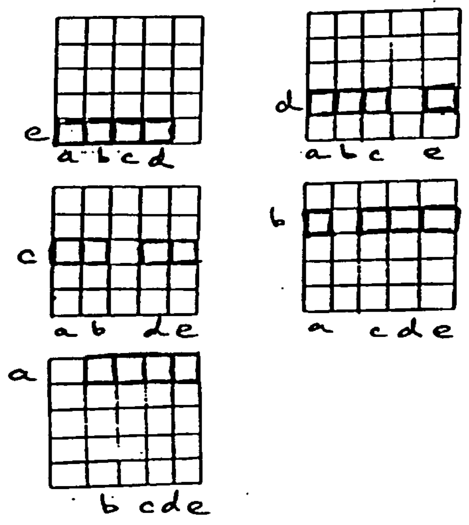
example:
m=3
n=5



DEFINING SUBARRAYS AND ~~COMPOSITE~~ COMPOSITE SUBMATRICES FOR m=3, n=5
SUBARRAY DIMENSIONS = $M \times (n-m)$ = 3×2
OF COMPOSITE SUBMATRICES = $n!/m!(n-m)! = 10$

FIG. 35

example:
m=4
n=5



DEFINING SUBARRAYS AND COMPOSITE SUBMATRICES FOR $\begin{matrix} m=4 \\ n=5 \end{matrix}$

SUBARRAY DIMENSIONS = $m \times (n-m) = 4 \times 1$

OF COMPOSITE SUBMATRICES = $n!/m!(n-m)! = 5$

FIG. 36

INTERNATIONAL SEARCH REPORT

International Application No
PCT/US 00/20398

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 H03M13/35 H03M13/47 G06F11/10 H04L1/00

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 H03M H04L G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

INSPEC, EPO-Internal, PAJ, WPI Data, IBM-TDB, COMPENDEX

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	A. COHEN: "Segmented Information Dispersal" 'Online! 1996 , UNIVERSITY OF CALIFORNIA , SAN DIEGO, USA XP002155787 Retrieved from the Internet: <URL: www.bell-labs.com/user/arielcohen> 'retrieved on 2000-12-18! page 1 -page 8, line 9 page 36 -page 39, line 4 page 45 -page 47, line 9 page 54 -page 55 ---	1-3

-/--

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search 20 December 2000	Date of mailing of the international search report 05/01/2001
Name and mailing address of the ISA European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Authorized officer Absalom, R

INTERNATIONAL SEARCH REPORT

Inte   al Application No
PCT/US 00/20398

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication,where appropriate, of the relevant passages	Relevant to claim No.
A	RABIN M O: "EFFICIENT DISPERSAL OF INFORMATION FOR SECURITY, LOAD BALANCING, AND FAULT TOLERANCE" JOURNAL OF THE ASSOCIATION FOR COMPUTING MACHINERY,XX,XX, vol. 36, no. 2, 1 April 1989 (1989-04-01), pages 335-348, XP000570108 ISSN: 0004-5411 the whole document ---	1-18
A	T J E SCHWARZ ET AL.: "Reliability and Performance of RAIDs" IEEE TRANSACTIONS ON MAGNETICS, vol. 31, no. 2, March 1995 (1995-03), pages 1161-1166, XP002155785 USA the whole document ---	1-18
A	G A ALVEREZ ET AL.: "Declustered Disk Array architectures with Optimal and Near-optimal Parallelism" 25TH ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 27 June 1998 (1998-06-27), pages 109-120, XP002155786 Barcelona, Spain the whole document ---	1-18
A	G A ALVAREZ ET AL: "TOLERATING MULTIPLE FAILURES IN RAID ARCHITECTURES WITH OPTIMAL STORAGE AND UNIFORM DECLUSTERING" COMPUTER ARCHITECTURE NEWS,US,ASSOCIATION FOR COMPUTING MACHINERY, NEW YORK, vol. 25, no. 2, 1 May 1997 (1997-05-01), pages 62-72, XP000656567 ISSN: 0163-5964 the whole document ---	1-18
A	PARK C -I: "EFFICIENT PLACEMENT OF PARITY AND DATA TO TOLERATE TWO DISK FAILURES IN DISK ARRAY SYSTEMS" IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS,US,IEEE INC, NEW YORK, vol. 6, no. 11, 1 November 1995 (1995-11-01), pages 1177-1184, XP000539599 ISSN: 1045-9219 the whole document ---	1-18
A	EP 0 654 736 A (HITACHI, LTD) 24 May 1995 (1995-05-24) column 11, line 24 - line 32; figure 4 --- -/--	10

INTERNATIONAL SEARCH REPORT

Inter al Application No
PCT/US 00/20398

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication,where appropriate, of the relevant passages	Relevant to claim No.
A	US 5 623 595 A (BAILEY WILLIAM) 22 April 1997 (1997-04-22) abstract -----	16

INTERNATIONAL SEARCH REPORT

ɔrmation on patent family members

Inter ıal Application No
PCT/US 00/20398

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
EP 654736 A	24-05-1995	JP 7141121 A	02-06-1995
		DE 69425222 D	17-08-2000
		EP 1006445 A	07-06-2000
		US 5564116 A	08-10-1996
		US 5751937 A	12-05-1998
US 5623595 A	22-04-1997	AU 3723495 A	19-04-1996
		CA 2176384 A	04-04-1996
		DE 19581103 C	10-12-1998
		DE 19581103 T	14-11-1996
		GB 2297855 A,B	14-08-1996
		WO 9610228 A	04-04-1996